Generierung interaktiver Animationen für den Übersetzerbau

Dissertation

Zur Erlangung des Grades eines
Doktors der Ingenieurwissenschaften (Dr.-Ing.)
der Naturwissenschaftlich-Technischen Fakultät I
der Universität des Saarlandes

von

Diplom-Informatiker
Andreas Kerren

Juli, 2002

Tag des Kolloquiums:	16. September 2002
Dekan:	Prof. Dr.-Ing. P. Slusallek
Gutachter:	Prof. Dr. R. Wilhelm Prof. Dr. H. Hagen
Vorsitzender:	Prof. Dr.-Ing. G. Weikum

Zusammenfassung

Der Einsatz generischer bzw. generativer Verfahren zur Entwicklung und Anwendung interaktiver Lehr- und Lernsoftware wurde in Industrie und Ausbildung bislang kaum untersucht. Diese Arbeit befaßt sich mit der Visualisierung bzw. Animation von Algorithmen und Programmen, insbesondere von komplexen Berechnungsmodellen im Übersetzerbau, sowie mit der Verwendung generierter Animationen in Lehr- und Lernsoftware. Eigenschaften klassischer und aktueller Algorithmenanimationssysteme werden hierzu untersucht und klassifiziert. Ausgehend von den Eigenschaften dieser Systeme wird eine Reifikationstechnik vorgestellt, die der Softwarevisualisierung vollkommen neue Möglichkeiten eröffnet, z. B. die Animationssteuerung der Besuchsfolge von Schleifen oder die Vermischung von post mortem- und live-Visualisierungen. Zusätzlich unterstützt diese Technik zahlreiche mächtige Konzepte, die in klassischen Systemen ebenfalls realisiert wurden, wie etwa die parallele Ausführung von Programmpunkten mitsamt der damit verbundenen Animationen.

Die Technik wurde in einem Framework, genannt Ganimal, implementiert. Es besteht aus der Animationsbeschreibungssprache Ganila mit zugehörigem Compiler und einer Laufzeitumgebung. Ganila erweitert die Programmiersprache Java um Animationsannotationen und ermöglicht so die Spezifikation komplexer, interaktiver Animationen. Mit Ganimal entwickelte Animationen wurden in eine Lehr- und Lernsoftware für ein Themengebiet des Übersetzerbaus integriert, die lokal und über das Internet verwendet werden kann. Im Übersetzerbau oftmals angewandte Techniken zur Generierung von Software wurden dazu genutzt, die in der Lernsoftware enthaltenen Animationen teilweise automatisch zu erzeugen. Dieser generative Ansatz wird in einen lerntheoretischen Rahmen eingeordnet und eine daraus resultierende, neue Form des explorativen Lernens propagiert. Eine Evaluation der Lernsoftware in Form von Lernexperimenten mit über 100 Probanden belegt ihre Lerneffizienz und schließt die Arbeit ab.

Abstract

The use of generic and generative methods for the developement and application of interactive educational software is a relatively unexplored area in industry and education. In this thesis novel concepts for visualization and animation of algorithms or programs are presented, especially the animation of complex computational models for compiler design, as well as the integration of the generated animations in educational software systems. Concepts of classical and recent algorithm animation systems are surveyed and classified. Based on these characteristics, a reification technique is introduced that provides novel possibilities in software visualization, such as animation control of loops or mixing live and post mortem visualization. This technique also supports numerous powerful concepts, that have previously been implemented by classical algorithm animation systems, e. g. parallel execution of program points and associated animations.

The technique has been implemented in a framework, called Ganimal, consisting of the animation description language Ganila, a compiler and a runtime environment. Ganila extends the programming language Java by animation annotations and it supports the specification of complex interactive animations. Animations developed with Ganimal have been embedded into a learning system for compiler design, which can be used locally or online. Advanced techniques for software generation as used in compiler construction have been applied to the automatic generation of animations contained in the learning system. This generative approach is positioned within the scope of several educational theories and as a result a new way of explorative learning is propagated. An evaluation of the learning system based on experiments with more than 100 participants proves its efficiency and concludes this work.

Ausführliche Zusammenfassung

Die Entwicklung interaktiver, multimedialer Lehr- und Lernsoftware hat während der letzten Jahre in Industrie und Ausbildung große Beachtung erfahren. Fördergelder in Millionenhöhe wurden landesweit für die Erforschung und Entwicklung neuer Medien in der Lehre bereitgestellt. Generische bzw. generative Ansätze in der Implementierung und Anwendung dieser Lehr- und Lernsoftware allerdings sind bisher kaum untersucht, geschweige denn realisiert worden. Meist werden Autorensysteme für deren Implementierung eingesetzt, aber auch vermehrt HTML-Editoren. Dabei liegen die Vorteile generischer und generativer Techniken u. a. in einem hohen Grad an Wiederverwendbarkeit von Systemteilen sowie in der Reduzierung hoher Entwicklungskosten.

Die vorliegende Arbeit befaßt sich mit der Visualisierung bzw. Animation von Algorithmen und Programmen, insbesondere von komplexen Berechnungsmodellen im Übersetzerbau. Berechnungsmodelle, wie etwa endliche Automaten oder abstrakte Maschinen, lassen sich über interaktive Animationen besonders anschaulich vermitteln, sei es im Frontalunterricht oder im Selbststudium. Weiterhin diskutiert diese Arbeit die Verwendung automatisch generierter Animationen in Lehr- und Lernsoftware für den Übersetzerbau.

Ausgehend von einer Gegenüberstellung der Eigenschaften klassischer und neuerer Algorithmenanimationssysteme wird eine Reifikationstechnik vorgestellt, die die Assoziation von Metainformationen mit jedem Programmpunkt des zu animierenden Algorithmus unterstützt. Zur Laufzeit der Animation bildet jeder Programmpunkt ein Objekt mit manipulierbaren Eigenschaften. Dieses Verfahren eröffnet der Softwarevisualisierung vollkommen neue Möglichkeiten:

- die Animationssteuerung der Besuchsfolge von Schleifen,
- die Vermischung von post mortem- und live-Visualisierungen,
- die Einbindung alternativer Interesting Events,
- die Auswahl alternativer Codeblöcke und
- die Überprüfung von Invarianten an Programmpunkten.

Jede dieser Eigenschaften kann zur Laufzeit der Algorithmenanimation geändert werden, indem der Anwender entsprechende Programmpunkteinstellungen modifiziert. Zusätzlich unterstützt die Reifikationstechnik zahlreiche mächtige Konzepte, die in klassischen Systemen ebenfalls realisiert wurden, wie z. B.

- Interesting Events,
- verschiedene Sichten auf den auszuführenden Algorithmus,
- parallele Ausführung von Programmpunkten mitsamt der damit verbundenen Animationen oder
- Haltepunkte.

Das Verfahren wurde in einem Java-basierten Framework, genannt Ganimal, implementiert. Es besteht aus der Animationsbeschreibungssprache Ganila mit zugehörigem Compiler und einer Laufzeitumgebung, für die der Compiler entsprechenden Programmcode generiert. Ganila erweitert die Programmiersprache Java um Animationsannotationen und ermöglicht die Spezifikation komplexer, interaktiver Animationen. Die Arbeit führt die verschiedenen Ganila-Sprachkonstrukte ein und motiviert anhand vieler Beispiele die Reifikation von Programmpunkten. Für die Übersetzung von Ganila-Programmen in Java-Quellcode sind eine Reihe von statischen Analysen notwendig. Nach einer Diskussion dieser Analysen werden Übersetzungsschemata angegeben, die den Übersetzungsprozeß für eine Auswahl an Sprachkonstrukten semiformal beschreiben. Von Seiten der Implementierung sind Analysephase und Codegenerierung gemäß dem sogenannten Besuchermuster (Visitor Pattern) strukturiert. Der generierte Java-Quellcode bildet zusammen mit der Ganimal-Laufzeitumgebung interaktive, multimediale Animationen des Eingabeprogramms. Die Laufzeitumgebung wird komponentenweise anhand eines weiteren Entwurfsmusters, MVC (Model/View/Controller) genannt, strukturiert und erläutert. Sie unterstützt den Animationsentwickler in der Programmierung eigener Sichten auf den Algorithmus und stellt ein Basispaket mit graphischen Primitiven sowie eine Anzahl vordefinierter Standardsichten zur Verfügung. Eine graphische Benutzerschnittstelle dient zur Steuerung der Animation und zur Modifikation der o. g. Programmpunkteinstellungen.

Die Anwendbarkeit des Frameworks wird anhand einer Beispielimplementierung der Animation des Heapsort-Algorithmus verdeutlicht. Mit Ganimal entwickelte Animationen wurden in eine Lehr- und Lernsoftware für ein Themengebiet des Übersetzerbaus integriert: Das Lernsystem GaniFA ist ein HTML-basiertes elektronisches Textbuch, das die Generierung endlicher Automaten zum Gegenstand hat. Es kann lokal oder über das Internet verwendet werden. Im Übersetzerbau oftmals angewandte Techniken zur Generierung von Software wurden dazu genutzt, die in diesem Lernsystem enthaltenen Animationen teilweise automatisch zu erzeugen. Es ist möglich, sowohl den Generierungsprozeß des über einen beliebigen regulären Ausdruck spezifizierten endlichen Automaten als auch das Akzeptanzverhalten des generierten Automaten für ein beliebiges Eingabewort zu animieren. Dieser generative Ansatz wird in einen lerntheoretischen Rahmen eingeordnet und eine daraus resultierende, neue Form des explorativen Lernens propagiert. Die Arbeit erläutert hierzu grundlegende Theorien des Lernens und ordnet exemplarisch einige Lernsysteme für den Übersetzerbau, einschließlich GaniFA, in vier verschiedene Explorationsstufen ein. Es wird gezeigt, wie der diskutierte generative Ansatz neue Übungsformen unterstützt.

Eine Evaluation der generativen Lernsoftware GaniFA in Form von Lernexperimenten mit über 100 Probanden schließt die Arbeit ab. Teilnehmer vier verschiedener Versuchsgruppen studierten die Theorie zur Generierung endlicher Automaten mit jeweils unterschiedlichen Lernmitteln: im Frontalunterricht, mit einem Lehrbuch sowie mit Hilfe zweier Lernsysteme mit und ohne generativen Teil. Die Ergebnisse belegen, daß GaniFA hinsichtlich der Lerneffizienz besser als der Frontalunterricht und im Vergleich mit dem Lehrbuch nahezu gleich gut abschnitt. Darüber hinaus empfanden die Probanden die Möglichkeit, Animationen über die eingegebenen regulären Ausdrücke selbst zu erzeugen und auf diese Weise nachzuvollziehen, wie der Generierungsprozeß endlicher Automaten funktioniert, als in hohem Maße motivationssteigernd. GaniFA wird in der frei verfügbaren Online-Version mehrmals täglich besucht.

Verschiedene Aspekte dieser Arbeit wurden in mehreren Artikeln, Buchbeiträgen und Tagungsbänden veröffentlicht [DK02a, DGK02, KS02, DKW01, DK01, DK00a, DK02b] sowie auf einer Reihe von internationalen Konferenzen und Workshops vorgestellt.

Extended Abstract

During the past years, the developement of interactive, multimedial learning software has become more and more relevant in industry and education. Research funds of several millions EUR were nationwide supplied for the research and development of new educational media. However, generic and generative approaches for the implementation and application of such educational software systems are neither in the center of actual research interests nor have they been realized frequently. In most cases commercial authoring systems are used for their implementation. Recently, these authoring systems are replaced by so-called HTML-editors. Important advantages of generic and generative techniques over authoring systems are the high level of system reusability and the reduction of development costs.

In this thesis novel concepts for visualization and animation of algorithms or programs are described, especially the animation of complex computational models for compiler design. Computational models, such as finite automata or abstract machines, can be clearly taught by using interactive animations in traditional instruction as well as in self-study. Furthermore, this work presents the integration of automatically generated animations into educational software systems for compiler design.

Based on a short survey of classical and recent algorithm animation systems, a reification technique is introduced that allows arbitrary meta-information to be associated with the program points of the underlying algorithm. During execution, each program point corresponds to an object whose meta-information can be manipulated. As a consequence, this technique provides novel possibilities in software visualization, such as

- the animation control of loops,
- the mixing of live and post mortem visualization,
- the specification of alternative interesting events,
- the choice of alternative code blocks, and
- the visualization of invariants for program points and blocks.

Each of these features can be manipulated at the runtime of algorithm animation by changing settings attached to each program point. This technique also supports numerous powerful concepts that have previously been implemented by classical algorithm animation systems, e. g.

- interesting events,
- several views on the algorithm,
- parallel execution of program points and associated animations or
- breakpoints.

The reification technique has been implemented in a framework, called Ganimal, consisting of the animation description language Ganila, a compiler, and a runtime environment, in which the compiled programm code is executed. Ganila extends the programming language Java by animation annotations and it supports the specification of complex, interactive animations. In this work, different constructs of the Ganila language are introduced and the reification of program points is motivated. To translate Ganila programs into Java source code, a number of static analyses are necessary. After a discussion of these analyses some code translation schemes are given, which describe the compilation process with a choice of Ganila constructs in a semi-formal way. Analysis phase and code generation of the implementation are structured in accordance with the so-called visitor pattern. The generated Java source code produces the interactive animations for the input program in combination with the runtime environment. The runtime environment, that is precisely described in this thesis, implements another design pattern, called MVC (Model/View/Controller). It supports the animation developer to implement user-definied views on the algorithm and offers a base package which consists of a set of primitive methods for graphical output as well as a number of predefined standard views. A graphical user interface can be used to control the animation and to manipulate the settings.

The applicability of Ganimal is demonstrated by an example implementation of an algorithm animation for Heapsort. A more complex example represents the embedding of animations developed with Ganimal into a learning system for compiler design: GaniFA is an HTML-based electronic textbook on the theory of generating finite automata. It can be used locally or online. In the area of compiler construction frequently applied techniques for software generation have been used for the automatic generation of animations the learning system contains. It is possible to visualize both the generation process of the finite automaton specified by a regular expression and the acceptance behaviour of the generated automaton on an arbitrary input word. This generative approach is discussed within the scope of several educational theories and as a result a new way of exploratory learning is propagated. To this end, seminal educational theories are explained and four levels of increasing exploration in compiler design are introduced. For each level one educational system is exemplified. The generative approach corresponds to the highest level of exploration and provides new kinds of excercises.

An evaluation of the generative learning system GaniFA based on experiments with more than 100 participants proves its efficiency and concludes this work. Four learner groups studied the theory of generating finite automata each with the help of different methods: classical instruction, text book and two learning systems with and without generative part. The results prove, that the learning efficiency of GaniFA is higher than

the one of classical instruction and nearly as good as the text book's. The participants esteemed the possibilities to enter regular expressions and to generate animations of finite automata in order to understand how finite automata work. They confirmed that motivation increased because of using this system. The system's freely available online version is usually visited several times a day.

Many aspects of this work have been published in various articles, book contributions and conference proceedings [DK02a, DGK02, KS02, DKW01, DK01, DK00a, DK02b] and were presented at a number of international conferences and workshops.

Danksagung

Mein Dank gilt Prof. Dr. Reinhard Wilhelm, der mir während der letzten vier Jahre an seinem Lehrstuhl ein ausgezeichnetes Umfeld für die Erstellung dieser Arbeit zur Verfügung stellte. Seine Kommentare und Ratschläge eröffneten mir viele neue Erkenntnisse und unterschiedliche Sichtweisen. Weiterhin gab er mir stets großen Freiraum und ermöglichte mir mehrere Reisen zu Workshops und Konferenzen im In- und Ausland, in denen ich meine Arbeiten vorstellen durfte. Als Zweitgutachter dieser Dissertation stellte sich Prof. Dr. Hans Hagen zur Verfügung, dem ich hiermit ebenfalls mein Dankeswort aussprechen möchte.

Ich möchte Dr. Stephan Diehl für seine Betreuung und Ratschläge danken, die er mir in den letzten Jahren angedeien ließ. Diese Arbeit, von der Konzeption über die Implementierungsphase bis hin zur Niederschrift, hat von diesen Momenten, seiner Einsicht und Erfahrung in weitem Maße profitiert. Mein herzlicher Dank gilt auch allen Studenten und Mitarbeitern des GANIMAL-Projekts. Insbesondere danke ich Beatrix Braune, die in der Anfangsphase des Projekts viele wertvolle Hinweise zur Gestaltung des GANIMAL-Frameworks gab. Gleiches gilt für Thomas Kunze, der weite Teile des graphischen Basispakets programmierte. Ich danke Carsten Görg für seine Implementierung der Layoutalgorithmen für Graphen, einschließlich der Entwicklung des Vorausschauenden Graphlayouts, sowie Torsten Weller für die Programmierung einiger Laufzeitsystemkomponenten und des GANIFA-Applets. Für die effiziente Zusammenarbeit hinsichtlich der Evaluation der in dieser Arbeit vorgestellten Lernsysteme bedanke ich mich neben den bereits genannten Personen ganz besonders bei Julia Kneer und Christoph Glasmacher. Beide leisteten zur Auswertung der erhobenen Daten einen sehr großen Beitrag.

Meine besondere Dankbarkeit geht an meine Partnerin, Ulla Huber, für sorgfältiges Korrekturlesen sowie für ihre Unterstützung, Nervenstärke und Geduld während der Arbeit an meiner Dissertation.

Andreas Kerren

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einleitung

In der Entwicklung von Lehr- und Lernsoftware spielen generische und generative Methoden immer noch eine sehr untergeordnete Rolle. Kommerzielle Produkte werden häufig von Lehrbuchverlagen angeboten. Die dort eingesetzten Programmierer benutzen für einen Großteil der Produktion von Lehr- und Lernsoftware weitverbreitete Autorensysteme wie Macromedia Director [Mac02a] oder Asymetrix ToolBook [Asy02]. In Ausnahmefällen wird auch C++ oder eine andere Hochsprache für die Programmierung verwendet. Darüber hinaus werden im Zuge der verstärkten Online-Aktivitäten der Verlage auch vermehrt HTML-Editoren, Java-Anwendungen und Macromedia Flash [Mac02b] eingesetzt (vgl. [KWD00]).

Allerdings sind die genannten Autorensysteme auch sehr komplex und erfordern teilweise eine gründliche Einarbeitung in die Bedienung und Programmierung meist eingebauter, systemspezifischer Scriptsprachen, wie etwa OpenScript bei ToolBook. Werden komplexere Lernsysteme verlangt, so kommt man mit der Programmierung durch diese eingebauten Scriptsprachen nicht weiter, weil sie meist keine Konzepte wie Modularität, Wiederverwendung von Komponenten, Objektorientierung o. ä. besitzen. Hier wird dann die Erstellung von Programmen notwendig, die in Hochsprachen wie C++, Java etc. programmiert wurden und über geeignete Schnittstellen mit der eigentlichen Autorensystemanwendung kommunizieren [Ker99].

Generische Lehr- und Lernsoftware wird für einen Unterrichtsbereich erstellt und dann für spezielle Lehr- bzw. Lerneinheiten instantiiert. Dazu sind in einem geplanten Lehr- und Lernsystem alle gemeinsamen Bestandteile sämtlicher Instanzen zu identifizieren. Zum Beispiel könnte die Lernzielkontrolle ein gemeinsamer Bestandteil von Systemen über viele Fächer hinweg sein. Die Benutzeroberfläche sowie die Schnittstellen zu Wörterbüchern oder anderen Wissensquellen sind eher fachspezifisch. Schließlich bleiben die Elemente übrig, die spezifisch für ein Teilgebiet eines Faches sind (vgl. [KWD00]). Das Lehr- und Lernsystem InterTalk [DO02] beispielsweise realisiert dieses Konzept.

Einen etwas weitergehenden Ansatz stellen *generative* Systeme dar, die Software automatisch aus Spezifikationen erzeugen. In Lehr- und Lernsystemen existieren z. B. oftmals die gleichen Arten von Übungen, d. h. dort bietet sich ein guter Ansatzpunkt für den Einsatz generativer Werkzeuge an. Nimmt man als Beispiel einen Lückentexteditor, dann könnte ein generatives System Wörter als Lücke markieren und interne Links zu einer entsprechenden Wortdatenbank setzen. Es ist ebenfalls denkbar, daß es Lücken automatisch erzeugt, indem es z. B. alle Akkusativfälle herausfiltert und durch Lücken ersetzt. In diesem Zusammenhang könnte auch eine Lernzielkontrolle miterzeugt werden. Die Vorteile generischer Lehr- und Lernsysteme bzw. generativer Werkzeuge werden im folgenden aufgezählt:

- Sie können von Lehrenden und von Lernenden gleichermaßen verwendet werden.
- Sie erhöhen den Grad an Wiederverwendbarkeit, da bestimmte Systemteile nicht von einem bestimmten Lerngebiet abhängen. Unter diesem Aspekt gewinnen generative und generische Techniken auch im Software-Engineering immer mehr an Bedeutung.
- Sie erleichtern aus demselben Grund die nachträgliche Beseitigung von Fehlern (Updates). Modifizierte Generatoren ermöglichen es zudem, Teile einer Lernsoftware neu zu erzeugen.
- Sie reduzieren damit Entwicklungskosten, die gerade im Bereich von multimedialer Lernsoftware sehr hoch sind. Kosten von über 60.000 € pro Unterrichtsstunde sind nicht ungewöhnlich (vgl. [Iss97]). Enthält eine Lernsoftware interaktive Visualisierungen und Animationen hoher Qualität, dann fallen die Produktionskosten noch höher aus.

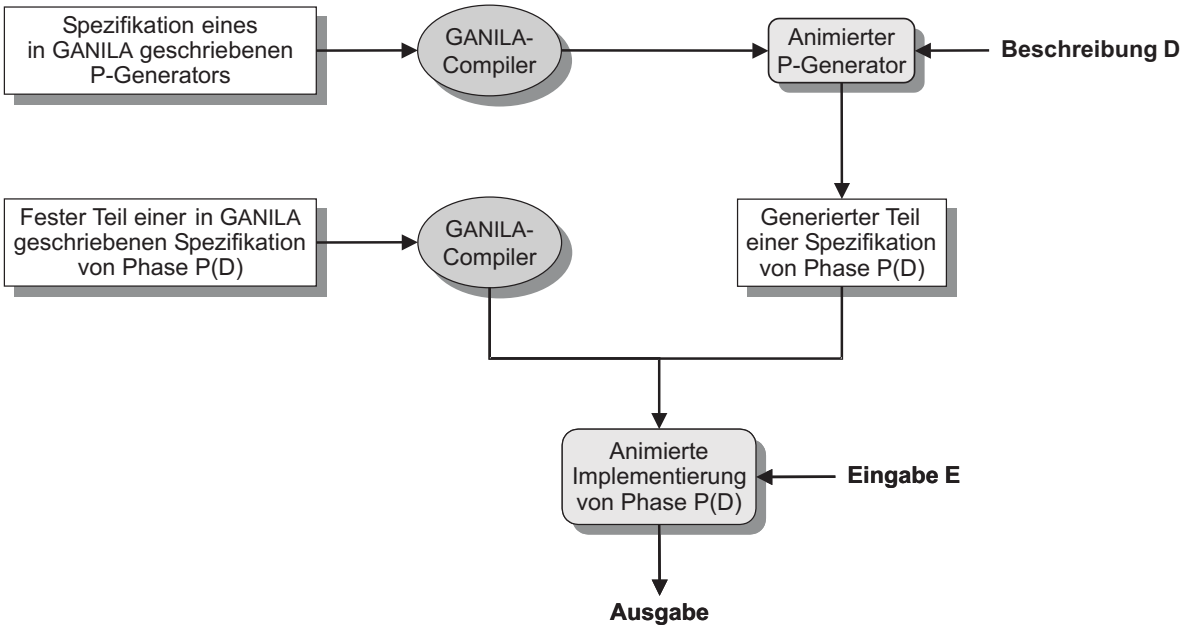
1.1 Generative Lernsoftware für den Übersetzerbau

Im Gebiet der Informatik, einschließlich des Übersetzerbaus, sind die zu vermittelnden Zusammenhänge und Algorithmen sehr abstrakt und in den meisten Fällen auch komplex. Man denke beispielsweise an Berechnungsmodelle wie endliche Automaten oder abstrakte Maschinen. Diese Lernthemen können mit Hilfe von Visualisierungen, insbesondere interaktiven Animationen, in der traditionellen Lehre und im Selbststudium wesentlich anschaulicher vermittelt werden [Wei97b, Sch00].

Algorithmenanimationssysteme sind Werkzeuge zur Produktion und Interaktion von bzw. mit graphischen Repräsentationen von Berechnungen. Solche Animationen unterstützen einerseits das Verständnis von Algorithmen und können somit als Lernhilfe eingesetzt werden, andererseits finden sie auch als Entwicklungswerkzeuge Verwendung. Im folgenden betrachten wir Algorithmenanimationen in Zusammenhang mit Lernsoftware für den Übersetzerbau. Dort spielt die Visualisierung von Generatoren eine wesentliche Rolle. Wir stellen diesbezüglich zwei unterschiedliche Ansätze vor, wobei der zweite Ansatz das Resultat dieser Arbeit widerspiegelt:

Ansatz I: Erweiterung existierender Generatoren

Eine etablierte Spezifikationssprache für eine Übersetzerphase, wie z. B. reguläre Ausdrücke für die lexikalische Analyse, wird um spezielle Animationsannotationen erweitert. Zusätzlich werden Komponenten des Berechnungszustands mit graphischen Strukturen (Fenster, Graphen, Textfelder, ...) assoziiert. Dann erzeugt ein erweiterter Generator aus einer so annotierten Spezifikation eine interaktive Animation der betrachteten Übersetzerphase. Der erste Ansatz erlaubt es nicht, den Generator selbst zu visualisieren. Der Generierungsprozeß wird nicht offenbart, und der Generator erscheint dem Lernenden (Lerner) als eine „Blackbox“.



Als Beispiel betrachten wir einen lexikalischen Analysatorgenerator. Der Markierung P entspricht die lexikalische Analyse, und D repräsentiert einen regulären Ausdruck. Der feste Teil ist der Treiber für die lexikalische Analyse, der generierte Teil ist eine Lookup-Tabelle, und die Implementierung von P(D) ist ein lexikalischer Analysator, auch Scanner genannt. Lernende können dann ein Wort E eingeben und das Akzeptanzverhalten des Scanners beobachten.

Abbildung 1.1: Entwicklung animierter Generatoren und Übersetzerphasen.

Ansatz II: Implementierung der Generatoren in einer Programmier- und Animationsbeschreibungssprache

Für den zweiten Ansatz machen wir von einer Eigenschaft Gebrauch, die viele Generatoren im Übersetzerbau haben: Sie erzeugen Tabellen, die zusammen mit einem festen Treiber die Implementierung einer Übersetzerphase ergeben. Wir können nun diese Eigenschaft für die Erzeugung der Visualisierungen der Generatoren und der durch sie generierten Übersetzungsphasen ausnutzen. Generatoren und Treiber werden in der Programmier- und Animationsbeschreibungssprache Ganila spezifiziert; eine Sprache, die Java um Animationsannotationen erweitert. Anschließend produziert ein Ganila-Compiler aus diesen Spezifikationen jeweils eine Implementierung des Generators bzw. des Treibers, wie in Abbildung 1.1 dargestellt ist. Zusammen mit einer Spezifikation der Übersetzerphase als Eingabe erzeugt die Generatorimplementierung die o. g. Tabelle und visualisiert diesen Prozeß über eine entsprechende Animation. Die Treiberimplementierung bildet mit der erzeugten Tabelle als Datenstruktur eine interaktive Animation der Übersetzerphase.

Ziel dieser Arbeit war die Entwicklung eines allgemeinen Rahmenwerks (Frameworks), das als technische Grundlage für die Realisierung des zweiten Ansatzes verwendet werden kann. Es besteht aus dem Ganila-Compiler und einer Laufzeitumgebung, für die der Compiler entsprechenden Programmcode generiert. Unter Anwendung dieses Frame-

works können Generatoren entwickelt werden, die interaktive, multimediale Visualisierungen und Animationen verschiedener Übersetzerphasen aber auch anderer Algorithmen erzeugen. Weiterhin wird die Animation der Generatoren selbst ermöglicht. Diese Animationen können als Bestandteil einer web-basierten Lernsoftware für den Übersetzerbau eingesetzt werden.

Generatoren für Visualisierungen und Animationen erfüllen in diesem Kontext zwei verschiedene Aufgaben: einerseits generieren sie aus vorgegebenen Spezifikationen Teile einer Lernsoftware, andererseits erlauben sie neue Übungsformen. Beschreibt das umschließende Hypermediadokument die verschiedenen Übersetzerphasen, dann kann es auch Übungen bereitstellen, die mit Hilfe der erzeugten Generatoren lösbar sind. Als Übungsaufgabe schreibt der Lernende Spezifikationen von Vorgängen oder Berechnungsmodellen. Wegen der Unentscheidbarkeit des Halteproblems ist jedoch eine automatische Überprüfung der Korrektheit für viele Aufgaben, die sich auf Berechnungsmodelle beziehen, nicht möglich. Daher werden in unserem Ansatz interaktive Animationen aus der eingegebenen Spezifikation des Lerners erzeugt, die er mit Hilfe eigener oder vorgegebener Beispiele überprüfen kann.

1.2 Das GANIMAL-Framework

Das in dieser Arbeit diskutierte Ganimal-Framework[1] wurde vollständig in der Programmiersprache Java implementiert. Dies betrifft sowohl den Ganila-Compiler (im folgenden GaJa genannt) als auch das Laufzeitsystem. Die Plattformunabhängigkeit von Java ermöglicht es, unabhängig vom lokal verfügbaren Computersystem, die mit dem Framework erstellten Visualisierungen in allen Gebieten der Lehre einzusetzen. Dabei kann sie lokal auf einem festen Rechner installiert oder über das WWW verwendet werden. Die mächtige Java-API (eine Klassenbibliothek) erleichtert u. a. die Behandlung von Benutzerinteraktion und die Entwicklung der graphischen Benutzerschnittstelle bzw. der graphischen Darstellung von Animationen.

Ganimal vereinigt Konzepte aus verschiedenen klassischen Algorithmenanimationssystemen: *Interesting Events* (IEs) und die Einbindung unterschiedlicher Sichten auf den zugrundeliegenden Algorithmus wurden von Balsa [BS84] übernommen. Dazu wird das Eingabeprogramm über spezielle Ganila-Konstrukte an wichtigen Punkten mit Interesting Events annotiert. Wann immer ein solches Event bei der Ausführung erreicht wird, sendet es Informationen über den aktuellen Programmzustand an die Sichten. Ganimal bietet eine Auswahl vordefinierter Standardsichten und unterstützt die Implementierung neuer Sichten durch den Entwickler einer Animation. Ebenso flossen die schrittweise Ausführung sowie Haltepunkte des Nachfolgers Balsa II [Bro88a] in das Entwicklungsdesign unseres Frameworks mit ein. Die in Ganimal mögliche parallele Ausführung von Programmpunkten wurde erstmals in Tango [Sta90a] realisiert.

[1] Ganimal – **G**enerierung interaktiver, multimedialer Visualisierungen und **Anima**tionen für **L**ernsoftware im Bereich des Übersetzerbaus. Das Ganimal-Projekt wurde von der Deutschen Forschungsgemeinschaft (DFG) unter Aktenzeichen WI 576/8-1 und WI 576/8-3 von Juli 1998 bis Dezember 2001 gefördert.

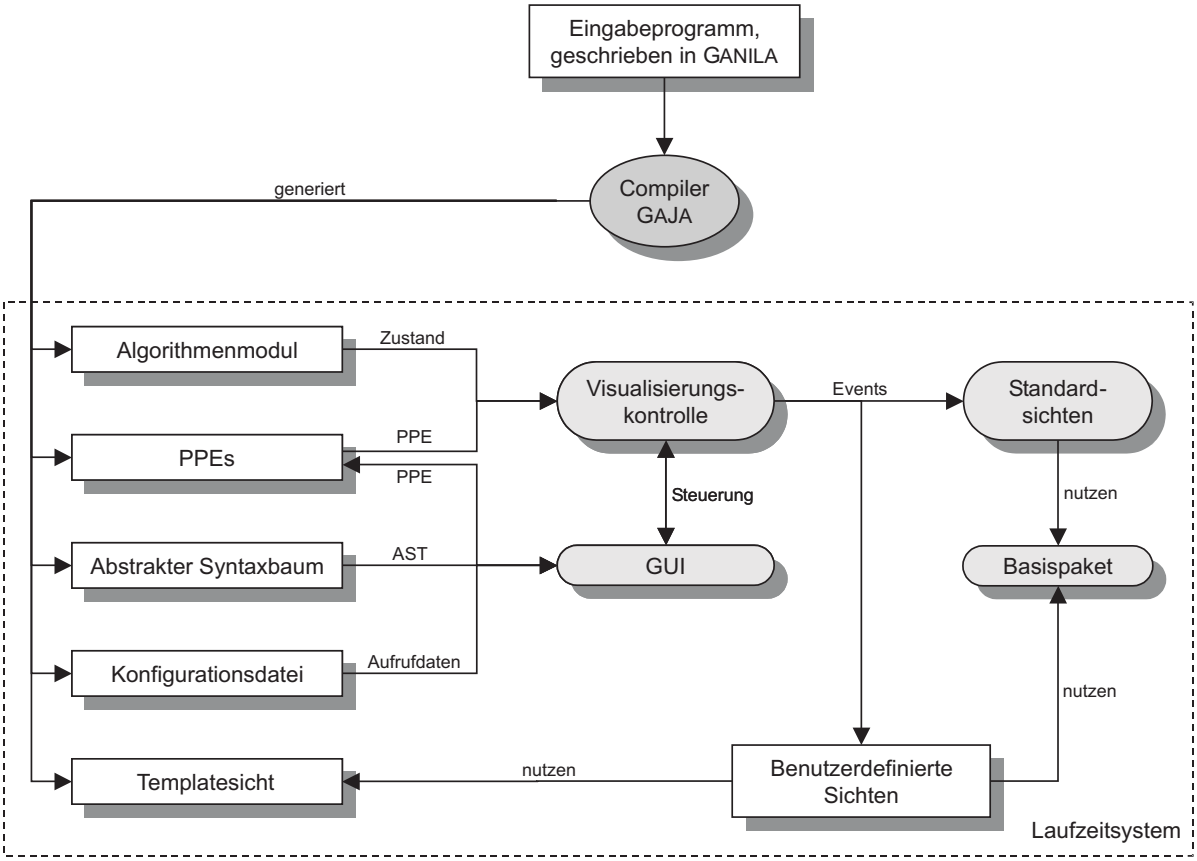


Abbildung 1.2: Das GANIMAL-Framework.

Darüber hinaus bietet GANIMAL auch neue Möglichkeiten wie etwa alternative Interesting Events, alternative Codeblöcke, Vermischung von post mortem- und live/online-Algorithmenanimationen, Animationssteuerung der Besuchsfolge von Schleifen sowie Überprüfung von Invarianten während der Ausführung von bestimmten Programmpunkten. Alle genannten Konzepte werden durch die Sprache GANILA unterstützt, die von GAJA in reines JAVA übersetzt wird. Der generierte Code erlaubt die Assoziation von Metainformationen mit jedem Programmpunkt des in GANILA implementierten Algorithmus. Eine graphische Benutzerschnittstelle unterstützt die Veränderung dieser Einstellungen zur Laufzeit einer Algorithmenanimation, z. B. wenn bestimmte Programmpunkte parallel oder sequentiell ausgeführt werden sollen. Dazu reifiziert der Compiler GAJA alle Programmpunkte: Jeder Programmpunkt bildet zur Laufzeit der Animation ein Objekt mit manipulierbaren Eigenschaften (*Programmpunkteinstellungen*, kurz PPEs).

Abbildung 1.2 stellt die Gesamtarchitektur des GANIMAL-Frameworks schematisch dar. Der Compiler GAJA erzeugt aus einem GANILA-Programm eine Reihe von Modulen, die in Kombination mit einer Laufzeitumgebung interaktive, multimediale Animationen bilden: das Algorithmenmodul entspricht dem reifizierten GANILA-Eingabeprogramm,

die PPEs repräsentieren initiale Metainformationen für die einzelnen Programmpunkte, der abstrakte Syntaxbaum des Eingabeprogramms wird zusammen mit einer Konfigurationsdatei in der graphischen Benutzerschnittstelle (Graphical User Interface – GUI) verwendet und die Templatesicht erleichtert die Implementierung von benutzerdefinierten Sichten.

Während der Ausführung der Algorithmenanimation sendet das Algorithmenmodul den aktuellen Zustand zu einer Visualisierungskontrolle. Falls der aktuelle Programmpunkt beispielsweise einem IE entspricht, entscheidet die Visualisierungskontrolle anhand der PPEs u. a. darüber, mit welchen Parametern das IE über einen Broadcast zu allen Sichten weitergeleitet wird. Jede Sicht überprüft nun wiederum selbst, ob sie einen Eventhandler für das IE aufrufen soll oder nicht. Abhängig vom aktuellen Animationsmodus produziert dieser Eventhandler graphische Ausgaben oder ändert lediglich den internen Zustand der Sicht. Sämtliche Sichten sollten das Basispaket verwenden, welches graphische Grundfunktionen enthält. Es besteht aus einer Menge von Java-Klassen mit primitiven Methoden für die Kommunikation sowie für das Zeichnen und die Animation graphischer Objekte. Die Benutzerschnittstelle kann nun verwendet werden, um die Einstellungen für jeden Programmpunkt zu ändern und die Ausführung des Algorithmus zu kontrollieren.

Mit diesen Eigenschaften erlaubt das Ganimal-Framework, auch klassische Algorithmenanimationen auf einfache Weise zu implementieren, z. B. Animationen von Sortierverfahren.

1.3 Aufbau der Arbeit

In Kapitel 2 wird eine Übersicht über Systeme zur Algorithmen- und Programmanimation gegeben. Dazu führen wir zunächst eine Reihe von Konzepten ein, die anhand klassischer Systeme erläutert werden. Anschließend werden neuere Systeme vorgestellt und nach bestimmten Designkriterien klassifiziert[2].

Kapitel 3 stellt die Animationsbeschreibungssprache Ganila vor und führt deren Sprachkonstrukte ein. Anhand von Beispielen wird die Reifikation von Programmpunkten im Detail motiviert[3].

Die Übersetzung von Ganila-Programmen in Java-Quellcode wird in Kapitel 4 präsentiert. Hier werden Codeerzeugungsschemata angegeben, die den Übersetzungsprozeß semiformal beschreiben. Der generierte Java-Quellcode bildet zusammen mit der Ganimal-Laufzeitumgebung interaktive, multimediale Animationen des Eingabeprogramms.

Kapitel 5 diskutiert die Laufzeitumgebung, deren Komponenten anhand eines speziellen Entwurfsmusters, MVC genannt, strukturiert werden.

Kapitel 6 zeigt die wichtigsten Anwendungen des Ganimal-Frameworks. Nach einem Beispielszenario, das dem Leser den typischen Entwicklungsprozeß einer Animation

[2]Die Systemübersicht basiert auf einer Buchkapiteleinleitung [KS02] mit J. T. Stasko als Koautor.

[3]Kapitel 3 ist eine erweiterte und aktualisierte Fassung von zwei Beiträgen [DK02a, DK02b], die der Verfasser gemeinsam mit S. Diehl veröffentlicht hat.

des Heapsort-Algorithmus verdeutlicht, beleuchten wir die Lernsoftware GaniFA: ein elektronisches Textbuch, welches die Generierung endlicher Automaten zum Gegenstand hat.

Um unseren generativen Ansatz zur Entwicklung von Animationen für den Übersetzerbau in einen didaktischen Rahmen einzuordnen, führt Kapitel 7 zunächst einige lerntheoretische Konzepte und Theorien ein. Auf Grundlage dieser Konzepte definieren wir ein eigenes Lernmodell für die Vermittlung von Berechnungsmodellen.

Eine Evaluation der Lernsoftware GaniFA mit mehr als 100 Studenten wird in Kapitel 8 beschrieben.

Kapitel 9 schließt diese Arbeit ab und gibt einen Ausblick auf zukünftige Entwicklungen.

Der Anhang enthält in einem ersten Teil den Ganila-Code sowie den daraus generierten Java-Code für die Animation des Heapsort-Algorithmus. Im zweiten Teil ist die deskriptive Statistik der durchgeführten Evaluation abgedruckt.

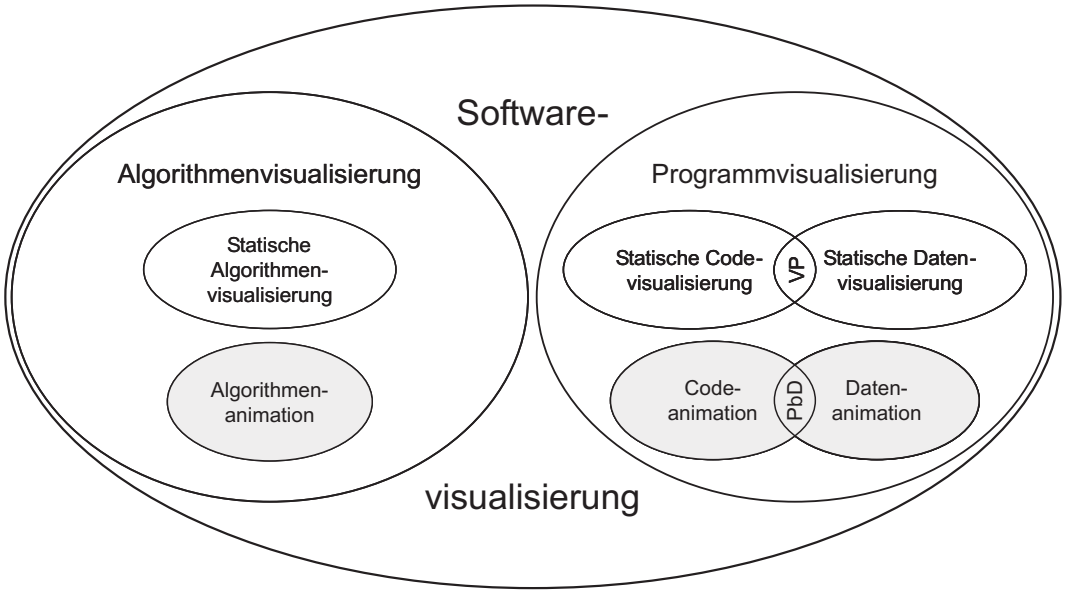
2 Verwandte Systeme zur Algorithmenanimation

Eine *Algorithmenanimation* visualisiert das Laufzeitverhalten eines Algorithmus, indem sie eine graphische Abstraktion seiner Daten und Operationen erzeugt. Dies geschieht dadurch, daß zunächst der aktuelle Zustand des Algorithmus auf eine graphische Repräsentation dieses Zustands abgebildet wird. Anschließend wird diese auf Basis der Operationen zwischen zwei aufeinanderfolgenden Berechnungszuständen animiert. Eine dynamische Visualisierung kann einerseits das bessere Verständnis der Arbeitsweise des Algorithmus unterstützen; andererseits deckt sie seine Stärken und Schwächen auf und erlaubt somit weitere Optimierungen.

Price et al. [PBS93] unterscheiden zwischen Algorithmenanimation und Programmanimation. Der erste Begriff bezieht sich auf eine dynamische Visualisierung einer Beschreibung von Software auf einer höheren Ebene (Algorithmus). Eine solche Beschreibung wird dann später in einer beliebigen Programmiersprache implementiert. Der zweite Begriff beschreibt dagegen die Verwendung von dynamischen Visualisierungstechniken für die Unterstützung des Verständnisses der eigentlichen Implementierungen von Programmen oder Datenstrukturen auf niedrigerer Ebene. Nach Price et al. sind die beiden betrachteten Gebiete gemeinsam ein Teil der *Softwarevisualisierung* (SV), wie in Abbildung 2.1 dargestellt und grau markiert ist. Die dort gezeigten statischen Visualisierungen (z. B. Flowcharts oder graphische Codedarstellung) sind nicht Gegenstand dieser Arbeit und werden in dieser Einführung nicht betrachtet. Ebenso werden wir die oben getroffene Unterscheidung nicht vornehmen, d. h. die nachfolgend diskutierten Systeme können sowohl unter dem Begriff der Algorithmenanimation als auch unter dem der Programmanimation subsummiert werden.

Einen Überblick zum Thema Softwarevisualisierung, insbesondere zu Systemen zur Programm- und Algorithmenanimation, geben zwei Sammelbände aus den Jahren 1996 und 1998 [EZ96, SDBP98]. Speziell der Band von Stasko et al. [SDBP98] enthält auch einige überarbeitete Versionen von grundlegenden Papieren über klassische Algorithmenanimationsysteme, Konzepte und Lernaspekte. Andere Veröffentlichungen geben Übersichten über verschiedene Aspekte der Algorithmenanimation, etwa über Taxonomien [Bro88b], den Gebrauch von Abstraktion [CR92] sowie über Benutzerschnittstellen [Glo98b].

In diesem Kapitel geben wir nur einen knappen Abriß über die historische Entwicklung der Softwarevisualisierung, wobei wir die Algorithmen- und Programmanimation besonders berücksichtigen und uns hier wiederum auf Systeme konzentrieren, die neue Konzepte eingeführt haben. Anschließend werden einige neuere Systeme anhand von vier Konzepten oder Dimensionen untersucht: Spezifikationstechnik, Visualisierungstech-



Das Venn-Diagramm zeigt die Zusammenhänge oft verwendeter Begriffe. Zu beachten ist, daß die beiden Schnittmengen VP (Visual Programming) und PbD (Programming by Demonstration) keine Teilmengen der Programmvisualisierung darstellen, sie haben mit dieser lediglich eine teilweise Überdeckung.

Abbildung 2.1: Klassifikation der Softwarevisualisierung (nach [PBS93]).

nik, Sprachparadigma und domänenspezifische Animation. Da diese Systeme erst in den letzten Jahren entwickelt worden sind, sind die meisten in den vorhergenannten Sammelbänden noch nicht enthalten. Aus Platzgründen beschreiben die nachfolgenden Abschnitte lediglich einige repräsentative Systeme und Ansätze. Eine umfassende Übersicht würde den Rahmen dieser Arbeit bei weitem sprengen.

2.1 Klassische Systeme und Konzepte

Knowltons Film [Kno66] über die Listenverarbeitung mit der Programmiersprache L6 gehörte zu den ersten Versuchen, Programmverhalten mit Hilfe von Animationstechniken zu visualisieren. Der Einsatz in der Lehre stand hierbei schon früh im Mittelpunkt [Bae73, Hop74], und eine Reihe weiterer Filme wurde produziert, insbesondere der Klassiker „Sorting Out Sorting“ [Bae81, Bae98], der neun verschiedene Sortierverfahren erläutert sowie ihre jeweiligen Laufzeiten illustriert.

Die Erfahrungen mit von Hand programmierten Algorithmenanimationen und die breite Verfügbarkeit graphikfähiger Rechner in den 80er Jahren führte in der Folge zur Entwicklung von Algorithmenanimationssystemen. Eines der ersten und in der Forschungsgemeinde der SV bekanntesten Systeme war Balsa [BS84, Bro87]. Es führte erstmals das Konzept der *Interesting Events* (IEs) ein und damit die Verwendung mehrerer Sichten (engl. Views) auf den gleichen Zustand. In diesem Ansatz wird das Programm durch den Visualisierer (eine Person, welche die Visualisierung spezifiziert) an wichtigen Punk-

ten mit IEs annotiert. Wann immer ein IE bei der Ausführung erreicht wird, sendet es Informationen über den aktuellen Programmzustand an die Sichten. Der Nachfolger Balsa II [Bro88a] wurde um Schritt- und Haltepunkte sowie eine Anzahl von anderen Funktionalitäten erweitert. In Zeus [Bro91], Cat [BN96] und dem späteren auf Java basierenden System Jcat [BR96] wurden die Sichten auf mehrere Rechner verteilt.

Das System Tango [Sta90a] implementierte das Pfad-Übergangs-Paradigma [Sta90b] und erlaubte damit gleichmäßige und nebenläufige Animationen von Zustandsübergängen. In seinem Nachfolger Polka [SK93] wurden diese Eigenschaften noch verbessert, um ein vereinfachtes Design von nebenläufigen Animationsaktionen zu ermöglichen (vgl. Abb. 2.2). Als ein Frontend zu Polka wurde ein interaktiver Animationsinterpreter geschaffen, genannt Samba [Sta97] (einschließlich des späteren Java-basierten Jsamba). Samba besteht aus einer Anzahl von parametrisierten Kommandos, die verschiedene Animationsaktionen unterstützen. Somit können Programme, die in einer beliebigen Programmiersprache geschrieben sind, einfach durch die textuelle Ausgabe dieser Kommandos an interessanten Stellen des Programms (IE-Programmpunkte) animiert werden. Diese werden post mortem von Samba eingelesen und die Animation wird schließlich ausgeführt.

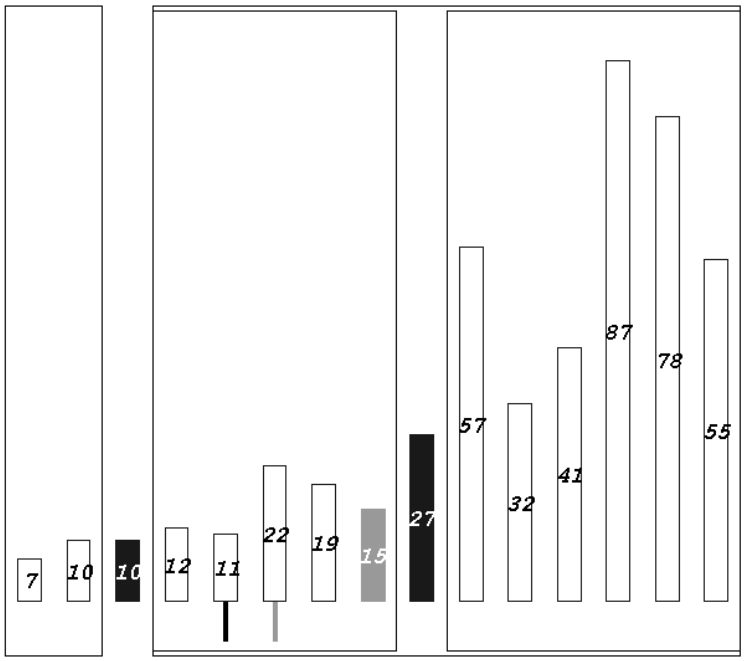


Abbildung 2.2: Polka-Animation des Quicksort-Algorithmus [SK93].

Das Pavane-System [RCWP92] verwirklichte erstmals ein deklaratives Animationsparadigma, in welchem Programmzustände ohne IEs unmittelbar auf Visualisierungszustände abgebildet wurden. Der Designer einer Animation kreierte zunächst eine initiale Zuordnung interessanter Datenstrukturen auf eine geeignete graphische Interpretation, etwa einen Variablenwert auf ein Rechteck entsprechender Höhe. Anschließend wurden

die entsprechenden Abbildungen im Programmlauf ausgeführt, so daß sich der Visualisierungszustand sukzessive mit dem Berechnungszustand änderte. Die Aufgabe des Systems war es also sicherzustellen, daß die graphische Interpretation zu jedem Zeitpunkt mit dem Berechnungszustand des zu animierenden Programms übereinstimmte.

Anstatt Algorithmen zu annotieren bzw. eine Abbildung von Programm- auf Visualisierungszustände zu definieren, wurde in GASP [TD95] der Weg beschritten, eine Bibliothek mit geometrischen Datentypen bereitzustellen, wobei die Operationen des Datentyps mit Animationen versehen sind. Damit erhält man dann die entsprechende Algorithmenanimation sozusagen als Nebeneffekt eines Programms, welches die Bibliothek verwendet.

Eine Anzahl weiterer beachtenswerter Algorithmenanimationssysteme und -werkzeuge wurde in den letzten zwei Jahrzehnten entwickelt. Einige der früheren Arbeiten auf diesem Gebiet seien im folgenden kurz genannt: Systeme in Smalltalk [LD85, Dui87], das ALADDIN-System [HR87, HHR89], das MOVIE-System [BK91], Animationen für Bücher über Algorithmen [Glo92, Glo98a] sowie das GAIGS-System [Nap90]. In den letzten Jahren wurden einige neue Systeme auf Workshops und Konferenzen präsentiert wie z. B. auf dem GI Workshop SV'2000 [DK00b], dem First International Program Visualization Workshop 2000 [Sut00] und dem Dagstuhl Seminar Software Visualization 2001 [Die02]. In den nächsten Abschnitten beschreiben wir einerseits kurz einige der neueren Systeme, andererseits bedeutende frühere Systeme im Hinblick auf vier wichtige Dimensionen: Spezifikationstechnik, Visualisierungstechnik, Sprachparadigma und Domänenspezifität.

2.2 Spezifikationstechnik

Eine wichtige praktische Aufgabe bei der Implementierung einer Algorithmenvisualisierung ist die Erstellung einer Spezifikation, welche beschreibt, wie die Visualisierung mit dem Algorithmus verbunden wird. SV-Forscher haben einige unterschiedliche Ansätze zur Lösung dieses Problems entwickelt. In diesem Abschnitt werden wir solche Ansätze untersuchen und Systeme zu deren Realisierung vorstellen. Zu beachten ist, daß einige Systeme in mehrere Kategorien von Spezifikationstechniken eingeordnet werden können.

2.2.1 Ereignisgetriebene Animation

Der Spezifikationsansatz über *Interesting Events* wurde erstmals durch das BALSA-System [BS84, Bro87] umgesetzt und in sehr vielen Algorithmenanimationssystemen, einschließlich dem BALSA-Nachfolger ZEUS [Bro91], verwendet. Wie bereits oben beschrieben, identifiziert der Visualisierer die Schlüsselstellen des Programms und annotiert diese mit IEs. Wann immer ein IE während der Ausführung des Programms erreicht wird, sendet das System ein parametrisiertes Event zu den verschiedenen Sichten. Animationen mit den in Abschnitt 2.1 vorgestellten Systemen TANGO und POLKA [SK93] werden auf die gleiche Art spezifiziert. Die IEs werden dort jedoch als *Algorithmenoperationen* bezeichnet.

Das ANIMAL-System [RSF00, RF02] verwendet ebenfalls diesen event-basierten Ansatz und unterstützt eine große Anzahl fortgeschrittener Features zur Algorithmenpräsentation, einschließlich einer dynamischen Abbildungsmöglichkeit von Programmauf Visualisierungszustände, Rückwärtsausführung, Internationalisierung des Animationsinhalts sowie flexible Import- und Exportfähigkeiten.

2.2.2 Zustandsgetriebene Animation

Einen alternativen Ansatz stellt die Spezifikation einer Abbildung zwischen Programm- und Visualisierungszustand dar, welche gewöhnlich durch den Visualisierer vor der Programmausführung definiert wird. Wie bereits oben diskutiert, hat sich das PAVANE-System dieser Technik [RC89, Rom98] bereits früh angenommen. Der Ansatz zustandsgetriebener Animationen wird ebenfalls von vielen neueren Systemen verwendet.

LEONARDO [DF99, DF01] ist eine integrierte Entwicklungsumgebung für die Erstellung, Ausführung und Animation von C-Programmen. Eine Visualisierung wird durch dem C-Programm hinzugefügte Deklarationen spezifiziert, die in einer deklarativen Sprache namens ALPHA geschrieben sind (siehe Abb. 2.3). LEONARDO unterstützt auch die vollständige Rückwärtsausführung einer Animation. Dies wird durch die Verwendung einer speziellen virtuellen Maschine ermöglicht, die das übersetzte C-Programm ausführt.

```
int i;

/**
  // We declare a window whose label is 1
  View(Out 1);

  // We declare a rectangle whose ID is 0, left-top
  // corner in (10,10), height 15, length always equal
  // to the content of program's variable i, and
  // belonging only to the view 1
  Rectangle(Out 0,Out 10,Out 10,Out L,Out 15,1)
    Assign L=i;

  // We declare that each rectangle having ID 0
  // in the view 1 must be cyan
  RectangleColor(0,Out Cyan,1);
**/

void main() { for (i=0; i<=100; i+=10); }
```

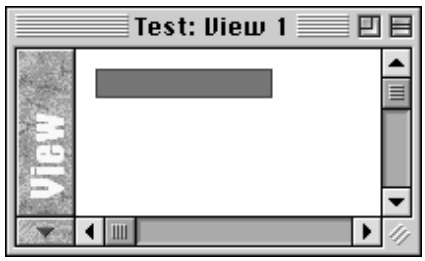


Abbildung 2.3: Deklarative Spezifikation in ALPHA und die sich ergebende Animation (vgl. [DF99]).

DAPHNIS ist ein auf Datenflußtracing basierendes Algorithmenanimationssystem. Einige Teilaspekte bei der Visualisierungsabstraktion werden vollautomatisch generiert. Für die Vorbereitung einer Animation ist ein externes Konfigurationsskript notwendig, das die graphische Repräsentation sowie eine Anzahl von Übersetzungsregeln für alle darzustellenden Variablen spezifiziert. Um die räumliche oder temporäre Unterdrückung unerwünschter Informationen zu erreichen, wird in diesem System eine besondere Art des Petrinetzformalismus angewendet, die den eigentlichen Prozeß der Programmausführung

beschreibt. Das DAPHNIS-System und sein zugrundeliegendes theoretisches Modell werden in [Fra02] ausführlich erläutert.

Demetrescu et al. stellen einen direkten Vergleich der beiden zuvor genannten Ansätze *Interesting Events* und *State Mapping* in [DFS02] vor. Sie diskutieren die Vor- und Nachteile dieser beiden Ansätze anhand mehrerer Szenarien: Die Benutzung von Interesting Events ist intuitiv und sehr gut geeignet, um Animationen speziellen und komplexen Bedürfnissen (z. B. verschiedenen Granularitäten) anzupassen. Dafür muß der Animationsentwickler aber meist viele zusätzliche Codezeilen implementieren. Der Ansatz zustandsgetriebener Animationen hat den Vorteil, diese mit sehr wenig Aufwand zu spezifizieren. Oft genügt nur eine zusätzliche Zeile Programmcode. Allerdings scheint dieser Ansatz im Vergleich zu Interesting Events nicht ganz so flexibel zu sein.

2.2.3 Visuelle Programmierung

Ein anderer Ansatz zur Spezifikation von Algorithmenanimationen ist der Gebrauch von Techniken aus der Visuellen Programmierung (VP). Diese Techniken stellen visuelle Notationen der verschiedenen Programmstrukturen und -anweisungen zur Verfügung, wodurch die Spezifikation von Programmen erleichtert werden soll. Das Ziel von VP ist also nicht die Erzeugung von Animationen, sondern von Anwendungssoftware. Als Ganzes betrachtet unterscheidet sich die VP daher von der SV [PBS93], aber die graphische Notation selbst stellt eine Art von statischer Code- bzw. Datenvisualisierung dar.

Die deklarative visuelle Programmiersprache FORMS/3 [CBC96] ist ein Projekt, in welchem Animationsfähigkeiten in eine visuelle Programmiersprache eingebettet wurden. Hierbei wird durch die Aufrechterhaltung eines Netzwerks von One-way-Constraints animiert. Ihre Entwickler haben eine erweiterte Version des Pfad-Übergangs-Paradigmas (S. 11) in die Sprache integriert und erhielten damit einen einzigartigen Ansatz zur Algorithmenanimation, insbesondere die nahtlose Integration von Animation in ihre Sprache. Die Erzeugung von Algorithmenanimationen kann somit als Nebeneffekt der Programmierung angesehen werden.

Ein mit der Visuellen Programmierung verwandtes Gebiet ist die Programmierung durch Demonstration (PbD). In PbD erläutert eine Person ein Beispiel oder eine Operation einer Aufgabe, und das PbD-System inferiert aus dieser Beschreibung ein Programm für die Aufgabe. DANCE [Sta91, Sta98] ist ein PbD-Interface für das TANGO-System. Nachdem der Benutzer ein Animationsszenario mit Hilfe eines graphischen Editors, der direkte Manipulation unterstützt, demonstriert hat, generiert das DANCE-System eine Reihe von Textzeilen, welche die Animation spezifizieren. Dieser Text wird dann als Eingabe für das TANGO-System verwendet.

2.2.4 Automatische Animation

Zumindest aus der Sicht des Algorithmenentwicklers ist die automatische Generierung einer Animation der wohl einfachste Weg, eine Algorithmenanimation zu spezifizieren. Allerdings ist die automatische Erzeugung von Algorithmenanimationen als Ganzes betrachtet extrem schwierig [Bro88b]. Daher unterstützen die diesem Ansatz folgenden

Systeme verschiedene Ebenen der Automation, um eine Algorithmenanimation zu beschreiben. Es liegt auf der Hand, daß diese Spezifikationsart besonders gut zur Fehlersuche geeignet ist [MS94], weil eine automatisierte Animation wenig oder keinen Aufwand von Seiten des Programmierers erfordert.

Ein frühes System, welches diesem Gebiet zugeordnet werden konnte, war UWPI [HWF90]. Es bot automatische Animationen durch die Nutzung eines kleinen Expertensystems an, welches die Visualisierung der Datenstrukturen und Operationen eines Algorithmus auswählte, in dem es versuchte, die im Programm verwendeten abstrakten Datentypen herauszufinden. Das System konnte Abstraktionen von Datenstrukturen höherer Ebene darstellen, auch wenn es diese selbst nicht wirklich „verstehen“ konnte.

JELIOT ist eine ganze Familie von Programmanimationsumgebungen. Einige Mitglieder dieser Familie unterstützen ein semi-automatisches Paradigma durch eine benutzerdefinierte visuelle Semantik von Programmstrukturen oder durch die Selektion der für eine Visualisierung am geeignetsten Strukturen. Eines der Systeme wurde ausschließlich für Programmieranfänger entwickelt und ermöglicht eine vollständig automatisierte Animation. Es erlaubt keine Anpassung der Animation durch den Benutzer. Ben-Ari et al. geben in [BMST02] eine Übersicht über die JELIOT-Familie und diskutieren einige empirische Evaluationen ausgewählter Teilsysteme.

Eine andere Technik, die thematisch in diese Kategorie paßt, ist die Verwendung spezieller Pseudocodesprachen, mit deren Hilfe Programmierer ihren Programmcode implementieren und die Animation ebenfalls automatisch produziert wird. ALGORITHMA 99 [CLK00] ist beispielsweise ein System, welches diesen Ansatz realisiert.

2.3 Visualisierungstechnik

Einer der wichtigsten Aufgaben in der SV ist die Gestaltung des graphischen Erscheinungsbildes einer Visualisierung oder Animation. In diesem Zusammenhang müssen vielfältige Problemfelder beachtet werden, etwa welche Information dargestellt werden soll, wie dies zu geschehen hat, ob es einen Fokus auf die wichtigsten Elemente geben soll usw. Brown und Hershberger geben zu diesem Thema eine gute Übersicht über fundamentale Techniken [BH98a]:

- Basistechniken
  - Verwendung mehrerer Sichten auf den Algorithmus.
  - Darstellung des Zustands.
  - Chronologischer Ablauf des Algorithmus (History).
  - Fließende vs. diskrete Übergänge.
  - Visualisierung mehrerer Algorithmen zur gleichen Zeit.
  - Auswahl der Eingabedaten.

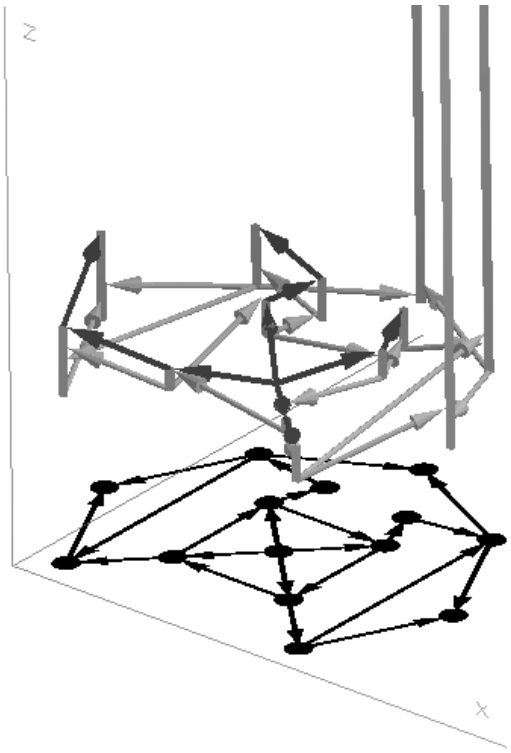


Abbildung 2.4: ZEUS3D-Animation des SSSP-Algorithmus (vgl. [BN93]).

- Färbungstechniken
  - Kodierung des Zustands der Datenstrukturen.
  - Hervorhebung von wichtigen Aktivitäten.
  - Identische Farbe für dieselbe Datenstruktur, die in mehreren Sichten dargestellt wird.
  - Verdeutlichung von wiederkehrenden Mustern.

An dieser Stelle beschränken wir uns auf die Diskussion von drei Aspekten der Algorithmenvisualisierung, die in letzter Zeit besondere Beachtung erfahren haben: 3D-Algorithmenanimation, Auralisierung und Webeinsatz.

2.3.1 3D-Algorithmenanimation

Es gibt mehrere Gründe dafür, 3D-Graphikfähigkeiten in ein Algorithmenanimationssystem zu integrieren. Die dritte Dimension kann dazu verwendet werden, den Zeitverlauf abzubilden (History), mehrere Sichten zu vereinen und zusätzliche Informationen darzustellen [BN98]. Die klassischen Systeme POLKA [SW93] und ZEUS [BN93] wurden um 3D-Fähigkeiten erweitert. Abbildung 2.4 zeigt eine 3D-Variante einer Animation des Kürzeste-Wege-Algorithmus (Single-Source-Shortest-Path, Abk. SSSP). Brown und

Najork integrierten ihre früheren Ergebnisse aus dem plattformabhängigen Werkzeug Zeus3D in das o. g. Jcat-System. Mit Hilfe des resultierenden, auf Java basierenden Systems können 3D-Animationen in jedem Standardwebbrowser gestartet werden [BN01, Naj01]. Diese wurden unter Verwendung der objektorientierten, Szenengraph-basierten Bibliothek Java3D (Plugin und API) entwickelt.

Ebenso erforschten Tal und Dobkin 3D-Animationen von geometrischen Algorithmen anhand ihres Gasp-Systems [TD95]. Sie schufen eine Bibliothek von geometrischen Datentypen einschließlich ihrer Operationen, welche mit Animationsinstruktionen versehen wurden.

2.3.2 Auralisierung

In der SV kann Audio dazu verwendet werden, visuelle Repräsentationen zu verstärken, zu ersetzen, Muster wiederzuerkennen, wichtige Ereignisse in einem Datenstrom zu identifizieren und Ausnahmebedingungen zu signalisieren [BH98b]. In diesem Zusammenhang wurde in den letzten Jahren die Abbildung von Informationen auf musikalische Klänge unter Berücksichtigung verschiedener Parameter wie Rhythmus, Harmonie, Melodie sowie speziellen Leitmotiven studiert.

Caitlin [VA96] ist ein Pascal-Präprozessor, der es einem Benutzer erlaubt, eine Auralisierung auf der Basis eines hierarchischen Leitmotivdesigns für jedes Programmkonstrukt (z. B. für eine `for`-Anweisung) zu spezifizieren. Caitlin unterstützt jedoch keine Datenauralisierung. Empirische Studien [VA00] dieses Systems zeigten, daß Programmieranfänger die musikalische Vertonung interpretieren konnten, daß musikalisches Vorwissen keinen signifikanten Effekt auf die Leistung hatte und daß eine musikalische Vertonung unter bestimmten Umständen für die Fehlersuche in Programmen als gute Hilfe angesehen werden konnte.

Das musikalische Datenvertonungswerkzeug Muse [LBH+97] unterstützt eine flexible Abbildung der Daten auf musikalischen Sound. Dabei können diese Daten aus einer beliebigen Quelle stammen. Das System wurde für die SGI-Plattform geschrieben und bietet verschiedene Abbildungsparameter von Daten auf Sound an, wie etwa Rhythmus, Tempo, Lautstärke, Tonlage und Harmonie.

Ein sehr ähnliches System ist Faust [WC97]. Es unterstützt eine einfache Abbildung der Programmdaten auf bestimmte Klangparameter durch manuell eingefügte IEs. Weiterhin kann der Programmierer die Soundsynthesealgorithmen wechseln bzw. den vordefinierten Synthesealgorithmen neue Eigenschaften und Attribute hinzufügen.

2.3.3 Webeinsatz

Mit dem wachsenden Gebrauch des World Wide Web als eine generische Anwendungs- und Darstellungsplattform sind viele neue Algorithmenanimationssysteme entwickelt worden, welche Animationen über das Web zur Verfügung stellen. Die bereits diskutierten Systeme JSamba und JCat sind zwei Beispiele. Weitere Systeme, die Animationen über das Web präsentieren, sind JHave [NEN00], Sort Animator [DB98], Jeliot [H+97] und Jawaa [PR98].

2.4 Sprachparadigmen

Verschiedene Sprachparadigmen verlangen nach unterschiedlichen Abstraktions- und Komponentenvisualisierungen. Dies ist durch ihre jeweilige Art der Berechnung und Problemlösungsmethode begründet. Der Übersichtsartikel von Oudshoorn, Widjaja und Ellershaw [OWE96] analysiert die Anforderungen an eine Visualisierung für eine Vielzahl von Programmierparadigmen und liefert eine einfache Taxonomie von Programmvisualisierungen. Im folgenden Abschnitt betrachten wir die wichtigsten Sprachparadigmen und stellen einige Beispielsysteme vor.

2.4.1 Imperative Programmiersprachen

Vom Standpunkt imperativer Programmiersprachen aus betrachtet, hat der Entwickler einer Algorithmenanimation Abstraktionen von Variablen, Datenstrukturen, Prozeduren bzw. von Funktionen und Kontrollstrukturen zu finden. Das o. g. Balsa-System [Bro88a] realisiert diesen Ansatz.

2.4.2 Funktionale Programmiersprachen

Die wichtigsten Abstraktionen für funktionale Sprachen sind Funktionen und Datenstrukturen. Ein textueller Browser zur Beobachtung des Auswertungsablaufs einer lazy-funktionalen Sprache wird in [WS97] diskutiert. Das System unterstützt die Navigation über das Ablaufprotokoll und kann sowohl als Debugger als auch als Lernhilfe dafür angesehen werden, wie Lazy-Auswertung funktioniert.

Das Kiel-System [Ber00] ist ein interaktives System für die Ausführung von first-order funktionalen Programmen, die in einer einfachen Untermenge von Standard ML geschrieben werden. Im Gegensatz zu dem vorher diskutierten System bietet Kiel einen Weg der graphischen Visualisierung des Auswertungsprozesses.

Foubister [Fou95] führt ein formales Modell für die Traversierung von graphischen Ablaufprotokollen von lazy-funktionalen Programmen ein. Das Modell stellt die visuelle Repräsentation der Graphreduktion durch Instantiierung graphischer Schablonen der verschiedenen Sprachkonstrukte dar. Es löst auch einige Probleme, die bei der graphischen Darstellung der Reduktion auftreten, z. B. Probleme bei sehr großen Graphen oder Planaritätsprobleme.

2.4.3 Objektorientierte Programmiersprachen

Das objektorientierte Paradigma hat einiges mit dem imperativen Sprachparadigma gemeinsam, d. h. die oben genannten Abstraktionen für imperative Sprachen gelten auch hier. Zusätzlich sind für Visualisierungen objektorientierter Programme typischerweise auch Abstraktionen von Objekten, Interobjektkommunikation eingeschlossen, von Bedeutung.

Noble [Nob02] beschäftigt sich mit Lösungen für die endemische Aliasproblematik innerhalb objektorientierter Programme: Ein bestimmtes Objekt kann von einer Anzahl

anderer Objekte über seine Adresse referenziert werden. Diese Tatsache könnte dem Algorithmenanimationssystem nicht bekannt sein und so zu Problemen bei der Animation führen. Das Papier diskutiert sowohl Programmanalysen zur Bestimmung der Ausdehnung des Aliasing als auch die Visualisierung von Eigentümerbäumen von Objekten in Java-Programmen.

Scene [KM96] produziert automatisch Szenariodiagramme (Eventtrace Diagrams) für existierende objektorientierte Systeme. Dieses Werkzeug unterstützt jedoch keine Visualisierung von Nachrichtenflüssen in einem objektorientierten Programm. Allerdings erlaubt es dem Benutzer, mit Hilfe eines „Active Text"-Frameworks durch mehrere Arten von assoziierten Dokumenten zu navigieren, z. B. durch Quellcode, Klassenschnittstellen oder -diagramme und Aufrufmatrizen.

2.4.4 Logische Programmiersprachen

Die interessanten Abstraktionen in logischen Programmen umfassen Klauseln und Unifikation. Eines der klassischen Systeme für die Visualisierung logischer Programme ist die Transparente Prolog Maschine Tpm [EB88]. Sie verwendet für den Suchraum das AND/OR-Baummodell. Die Ausführung des logischen Programms wird als Tiefensuche dargestellt. Ein anderes System, das logische Programme und ihre Funktionalität visualisiert, wird von Senay und Lazzeri [SL91] diskutiert. Es repräsentiert sowohl den statischen Quellcode als auch das dynamische Laufzeitverhalten des Eingabeprogramms anhand von zyklischen AND/OR-Graphen. Zyklen entstehen hier durch rekursive Definitionen von Relationen. Um das Verständnis eines Programmlaufs zu verbessern, erweitert das System den Graphen um Zeichen, die z. B. Unifikationen symbolisieren, sowie um zusätzliche Abhängigkeitsgraphen für Variablenbindungen, die zeigen, wie und wann Variablenwerte zur Laufzeit erzeugt werden.

2.5 Domänenspezifische Animationen

Die Suche nach adäquaten Abstraktionen von spezifischen Eigenschaften eines bestimmten Anwendungsgebiets (Domäne), etwa Eigenschaften von Realzeitalgorithmen, kann eine große Herausforderung darstellen, wenn man Algorithmen innerhalb einer solchen Domäne animieren möchte. Um die Abstraktionen zu bestimmen, ist ein besonderes Wissen über Objekttypen und -operationen notwendig, die in der speziellen Domäne überwiegen. Oftmals können auch allgemeine Algorithmenanimationssysteme verwendet werden, um domänenspezifische Animationen zu erstellen, aber der Aufwand kann höher und weitgreifender sein, als wenn man ein für die besondere Domäne spezialisiertes System nutzt.

Tal präsentiert in einem kürzlich erschienen Buchbeitrag [Tal02] ein konzeptionelles Modell für die Entwicklung domänenspezifischer Algorithmenanimationssysteme. Sie beleuchtet auch die praktische Implementierung dieses Modells anhand einiger Beispielsysteme, z. B. anhand des o. g. Gasp-Systems für die algorithmische Geometrie.

2.5.1 Algorithmische Geometrie

Im Anwendungsgebiet der algorithmischen Geometrie ist die Suche nach geeigneten Abstraktionen der Daten relativ einfach, wenn die Programmdaten positionale Informationen bereits enthalten. Diese können ohne komplizierte Transformationen bzw. ohne Interpretation der Programmdaten unmittelbar dargestellt werden.

Ein generisches Werkzeug für die interaktive Visualisierung von geometrischen Algorithmen ist GeoWin, welches in einem Artikel von Bäsken and Näher [BN02] beschrieben wird. Es handelt sich um einen C++-Datentyp, der mit algorithmischen Softwarebibliotheken wie LEDA verbunden werden kann.

Das Evega-System [KH01] ist eine auf Java basierende Visualisierungsumgebung für Graphalgorithmen und bietet eine große Anzahl an Features, um Graphen zu kreieren bzw. zu editieren, Visualisierungen darzustellen und verschiedene Algorithmen zu vergleichen. Zusätzlich unterstützt das System eine relativ einfache Implementierung neuer Algorithmen durch Klassenvererbung.

2.5.2 Nebenläufige Programme

Die Animation nebenläufiger Programme, welche üblicherweise sehr groß und komplex sind, sieht sich mit vielen Problemen hinsichtlich der Datenbeschaffung und -darstellung sowie der Programmausführung konfrontiert. Einige Problemkreise sind bereits inhärent visuell, z. B. korrespondiert ein Zyklus in einem Ressourcenzuteilungsgraphen mit einer Deadlocksituation. Ein anderes typisches Problem ist das nichtdeterministische Auftreten von Fehlern während der Programmausführung, deren Lokalisierung eine sehr geschickte Visualisierung des nebenläufigen Programms erfordert. Kraemer [Kra98] gibt eine gute Übersicht von Systemen für die Visualisierung nebenläufiger Programme.

Die event-basierte Parade-Umgebung [Sta95] unterstützt die Gestaltung und Implementierung von Animationen von parallelen und verteilten Programmen. Interesting Events können sowohl über Programmaufrufe und durch Kanäle empfangen als auch aus einer Datei gelesen werden, wie es auch mit dem o. g. Samba-Interpreter möglich ist. Eine besondere Komponente des Systems erfaßt die Events eines jeden Prozessors oder Prozesses und ermöglicht so dem Benutzer, die Ordnung dieser Events z. B. chronologisch oder logisch zu manipulieren. Das Polka-Animationssystem wird in Parade dazu verwendet, die graphischen Sichten zu erzeugen (vgl. Abb. 2.5).

Vade [MPT98] ist ein Client/Server-basiertes System für die Visualisierung von Algorithmen in einer verteilten Umgebung. Es umfaßt Bibliotheken, die die automatische Generierung von Viusalisierungen erleichtern, unterstützt die Kreierung von Webseiten für die Präsentation der fertiggestellten Animation und bietet Synchronisationsmethoden, welche die Konsistenz aufrechterhalten.

2.5.3 Realzeitanimationen

Einige Domänen, z. B. Netzwerkprotokolle, verlangen nach exakten Zeitbeziehungen im zugrundeliegenden Programm oder Algorithmus. Polka-Rc [SM95] ist eine Erweite-

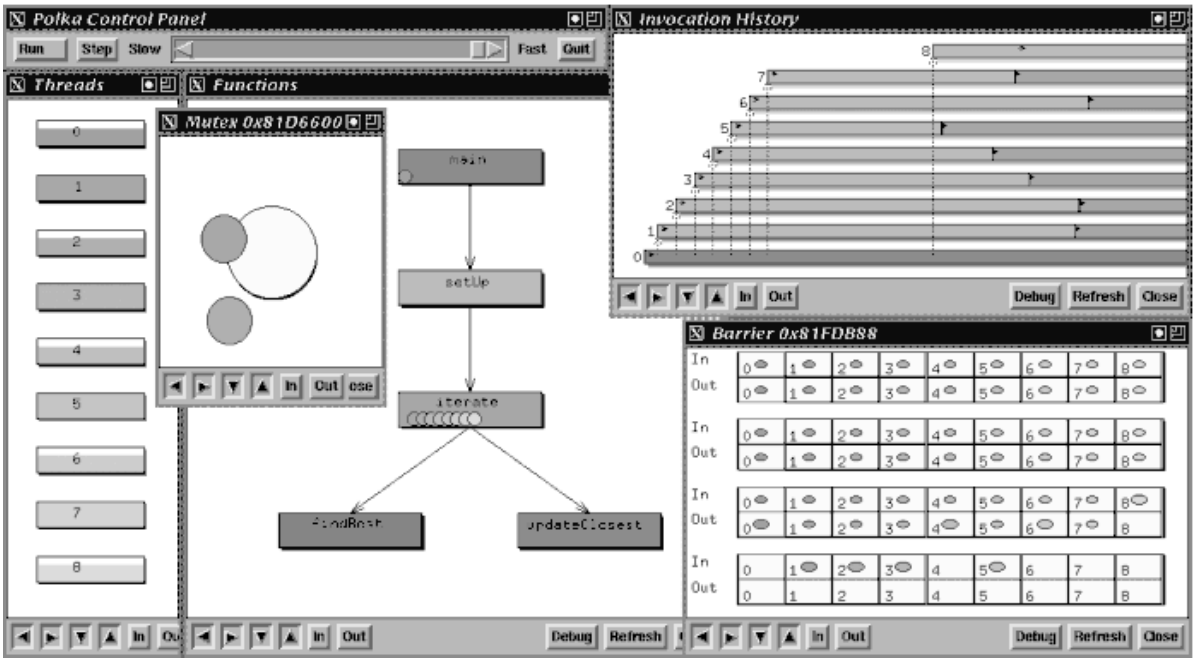


Abbildung 2.5: PARADE-Animation eines Programms mit Threads [Sta95].

rung von POLKA um Features für Realzeitanimationen, z. B. Animationsaktionen mit exakter Zeiteinteilung, festgelegtem Anfang und bestimmter Dauer. POLKA-RC bietet weiterhin eine flexible Mehrfachprozeßabbildung zwischen Programm und Visualisierung, d. h. das Programm und seine Animation laufen als separate Prozesse und kommunizieren über Sockets.

Das JOTSA-System [Rob98] ist ein JAVA-Paket für die Ausführung interaktiver webbasierter Algorithmenanimationen (einschließlich Netzwerkprotokolle). Mit JOTSA erstellte Animationen laufen auf jeder Plattform mit der gleichen Geschwindigkeit, vorausgesetzt die Hardware kann eine bestimmte Mindestframerate berechnen. Der Benutzer kann einen Faktor angeben, der das Verhältnis zwischen virtueller Systemzeit und Realzeit festlegt. Animationen können so verlangsamt werden, ohne zeitliche Abhängigkeiten zu beeinflussen. Das System unterstützt mehrfach unabhängig synchronisierte Sichten, Schwenken, Zoomen und Verbindungen von Objektgruppierungen. Zusätzlich weist es Funktionalitäten für die Animation von benutzerdefinierten event- und zeitgetriebenen Simulationen von Netzwerkprotokollen auf.

2.5.4 Berechnungsmodelle

Ein weiteres interessantes Gebiet ist die Animation von Berechnungsmodellen von formalen Sprachen, Mengen, Relationen und Funktionen. Diese Modelle sind eine Besonderheit des mathematischen Schlußfolgerns, aber typischerweise nicht zum Programmieren echter Hardware oder echter Anwendungen geeignet.

JFLAP [GR99] ist ein Werkzeug zum Erstellen und Simulieren mehrerer Arten von Automaten, z. B. von endlichen Automaten, Kellerautomaten und Turing-Maschinen. Dieses System wurde in JAVA implementiert und kann sowohl im konventionellen Unterricht als auch im Selbststudium verwendet werden.

2.5.5 Animation von Beweisen

Die Visualisierung von Beweisen in der theoretischen Informatik ist ein relativ unerforschtes Gebiet. Eines der wenigen Beispiele ist SCAPA [Pap99, PS97], ein System zur Animation von Beweisen der theoretischen Informatik. Das System generiert mit Hilfe eines Konverters aus einem in LaTeX verfaßten Beweis ein HTML-Dokument und eine JAVA-Datei. Der Visualisierer muß diese Dateien erweitern bzw. seinen Ansprüchen gemäß modifizieren. Die Beweisanimation wird anschließend unter Verwendung einer Erweiterung des LAMBADA-Werkzeugs vorgenommen. LAMBADA ist eine JAVA-basierte Reimplementierung des o. g. SAMBA-Systems.

3 Die Animationsbeschreibungssprache GANILA

Die Programmier- und Animationsbeschreibungssprache Ganila basiert auf der objektorientierten Programmiersprache Java und bietet eine Reihe von zusätzlichen Funktionalitäten. Einige der in Kapitel 2 eingeführten Konzepte verschiedener klassischer Algorithmenanimationssysteme wurden in die Sprache integriert. Dazu gehören Interesting Events und die Möglichkeit, verschiedene Sichten einzubinden (Balsa, [BS84]), schrittweise Ausführung und Haltepunkte (Balsa II, [Bro88a]) sowie die parallele Ausführung von Programmpunkten (Tango, [Sta90a]).

Zusätzlich bietet Ganila einige neue Konzepte wie alternative IEs und alternative Codeblöcke, Unterstützung von Vorausschauendem Graphlayout, Mischung von post mortem- und live/online-Algorithmenanimationen, kontrollierte Visualisierung von Schleifen bzw. Rekursionen und die Visualisierung von Invarianten für Programmpunkte und Blöcke [DGK02]. Viele dieser Eigenschaften können zur Laufzeit der Algorithmenanimation über eine graphische Benutzerschnittstelle ein- bzw. ausgeschaltet oder geändert werden. Beispielsweise kann ein Benutzer während der Laufzeit bestimmen, daß ein Programmblock alternativ zu einem anderen Block ausgeführt wird.

Die Sprache Ganila wird durch den Compiler GaJa in reines Java übersetzt. Die von GaJa generierten Programmodule produzieren in Kombination mit der in Kapitel 5 beschriebenen Laufzeitumgebung interaktive Animationen des Ganila-Programms. Die o. g. Eigenschaften lassen sich über reifizierte (wiederverkörperte) Programmpunkte realisieren, d. h. jeder im Eingabeprogramm vorhandene Programmpunkt erscheint zur Laufzeit des generierten und mit einem Java-Compiler übersetzten Ausgabeprogramms als ein manipulierbares Objekt mit bestimmten Eigenschaften.

Aufgrund des universellen Sprachdesigns können beliebige Algorithmen in Ganila implementiert und visualisiert werden. Andererseits ist es auch denkbar, reine Java-Programme von GaJa übersetzen und alle Programmpunkte reifizieren zu lassen. Man erhält damit zwar keine Visualisierung oder Animation, kann jedoch über die GUI den Ablauf des Programms verfolgen, Schritte ausführen oder die Geschwindigkeit regeln. Durch die Transformationen des Compilers GaJa kann das Programm zusammen mit einer Darstellung der Werte von Variablen etc. nach Fehlern abgesucht werden. Diese Möglichkeit war jedoch nicht Gegenstand unserer Forschung bzw. Entwicklungen und wird im weiteren nicht betrachtet.

Wir konzentrieren uns zunächst auf eine Beschreibung der Sprache Ganila und gehen auf deren verschiedene Konstrukte detailliert ein. Anschließend diskutiert Abschnitt 3.2

die Transformationen, welche zur Reifikation von Programmpunkten notwendig sind — wie der Compiler GaJa diese Transformationen und die Generierung der entsprechenden Java-Dateien durchführt, wird in Kapitel 4 erläutert — und verdeutlicht diese exemplarisch an ausgewählten Kontrollstrukturen und Ganila-Konstrukten. Abschnitt 3.3 schließt dieses Kapitel mit einer kurzen Zusammenfassung ab.

3.1 Sprachbeschreibung

Die Syntax von Ganila ist im wesentlichen eine Erweiterung der originalen Java-Syntax. Abbildung 3.1 zeigt einen Ausschnitt der Grammatik von Ganila in BNF-Form. Der überwiegende Teil dieser Grammatik wurde hier der Übersicht halber weggelassen. Sie entspricht größtenteils der kontextfreien Grammatik, die in der Sprachspezifikation der Programmiersprache Java [GJSB00] abgedruckt wurde.

ViewDecl	→	**view** <Identifier> **(** *ArgumentList* **) ;**
GanilaClassDecl	→	**ganila** *ClassDecl*
InteractiveMethodDecl	→	**interactive** *MethodDecl*
WhileStat	→	**while (** *Expr* **)** [**[** *CondExpr* **]**] *Stat*
DoStat	→	**do** *Stat* **while (** *Expr* **)** [**[** *CondExpr* **]**]
ForStat	→	**for (** [*ForInit*] **;** [*Expr*] **;** [*ForUpdate*] **)** [**[** *CondExpr* **]**] *Stat*
GanilaStat	→	***RECORD** | ***REPLAY** | ***BREAK** | *InterestingEvent* | *InvariantOp* | *ParallelOp* | *FoldingOp* | *AlternativeOp*
InterestingEvent	→	***IE_**<Identifier> **(** *ArgumentList* **) ;**
InvariantOp	→	***IV (** *CondExpr* **) *{** (*BlockStat*)* ***}**
ParallelOp	→	***{** (*BlockStat*)* ***} *II *{** (*BlockStat*)* ***}**
FoldingOp	→	***{** (*BlockStat*)* ***} *FOLD *{** (*InterestingEvent*)* ***}**
AlternativeOp	→	***{** (*BlockStat*)* ***} *ALT *{** (*BlockStat*)* ***}**

Abbildung 3.1: Ausschnitt einer Grammatik für die wichtigsten Ganila-Konstrukte in BNF-Form.

Die ersten drei Produktionen beschreiben Deklarationen, die außerhalb der Definition eines Klassenmethodenrumpfes auftauchen; die restlichen Produktionsregeln erzeugen zusätzlich zur originalen Java-Syntax eine Reihe neuer Anweisungen und erweitern die ursprünglichen Schleifenstrukturen um die Angabe von Bedingungen. Die meisten Erweiterungen beginnen zwecks besserer Unterscheidbarkeit von der Java-Syntax mit dem Sonderzeichen *****. Infolge dieser strikten optischen Trennung zwischen Java- und Ganila-Schlüsselwörtern — eine Ausnahme bilden die syntaktischen Erweiterungen der Kontrollstrukturen **for**, **do**, **while** — sprechen wir in dieser Arbeit oftmals vereinfachend von *Annotationen*, wenn sprachspezifische Ganila-Konstrukte gemeint sind.

3.1.1 Semantik der GANILA-Konstrukte

Im folgenden Abschnitt werden diese Annotationen detailliert erläutert, und ihre Anwendung wird anhand kleiner Codebeispiele verdeutlicht. Eine komplexere Implementierung des Heapsort-Algorithmus in der Sprache Ganila ist auf den Seiten 111 ff. zu finden.

Sichten

Ganila und das Basispaket stellen eine Menge von vordefinierten Standardsichten (siehe Abschnitt 5.4 auf Seite 88) zur Verfügung, die durch eine Sichtendeklaration **view** <*Name:* >(<*Parameterliste*>) in das Programm eingebunden werden:

```
// Eine Sicht ohne Parameter
view CodeView();

// Eine Sicht mit Parameter
view SoundView("http://www.cs.uni-sb.de/sounds/", new String[] { "beep.wav" });

// Eine algorithmenspezifische Sicht
view HeapsortTreeView();
```

Diese Deklarationen sind nach den üblichen Importdeklarationen von Java anzugeben, die mit dem Schlüsselwort **import** gekennzeichnet sind, aber noch vor der eigentlichen Klassendefinition des zu animierenden Algorithmus, die mit dem Ganila-Token **ganila** markiert ist. Hinter dem Schlüsselwort **view** folgt der Name der Sicht und eine möglicherweise leere Argumentliste zur Initialisierung. Mit den Argumenten können verschiedene Eigenschaften (Farben, Größe, URL etc.) der einzubindenden Sicht genau spezifiziert werden. Der Entwickler einer Animation kann die Standardsichten deklarieren und nutzen oder deren Funktionalität durch Vererbung neuen Bedürfnissen anpassen. Im oben gezeigten Beispiel ist eine neu kreierte Sicht HeapsortTreeView deklariert, die etwa von der vordefinierten GraphView erben könnte.

Interesting Events

Eine Teilmenge der Methoden, die in einer Sicht implementiert werden müssen, sind Eventhandler von Interesting Events. Das nachfolgende Programmfragment zeigt den Ganila-Code für eine einfache Operation, die oft exemplarisch in Animationen bekannter Algorithmenanimationssysteme enthalten ist: das Vertauschen der Inhalte zweier Feldelemente, hier A[i] und A[j].

```
help = A[i];   *IE_MoveToTemporary(A, i);
A[i] = A[j];   *IE_MoveElement(A, i, j);
A[j] = help;   *IE_MoveFromTemporary(A, j);
```

Interesting Events haben den Präfix ***IE_** und übertragen lokale Informationen über ihre Argumente zu den einzelnen Sichten, die diese Events bearbeiten. Es ist leicht zu sehen,

daß die Implementierung einer graphischen Sicht in unserem Beispiel den Wert des Feldelements A[i] zu einer Darstellung der Hilfsvariablen bewegen könnte. Daraufhin würde sich der entsprechende Wert von A[j] zu einer Repräsentation von A[i] und schließlich der Wert der Hilfsvariablen zu einer Repräsentation von A[j] bewegen. GANILA-Events (genauer: GEvent-Objekte) entsprechen fast genau den klassischen IEs anderer eventbasierter Algorithmenanimationssysteme wie ZEUS oder POLKA. Der entscheidende Unterschied ist, daß ein solches GEvent-Objekt zuerst zur zentralen Visualisierungskontrolle des GANIMAL-Laufzeitsystems gesendet wird und dabei einem gewöhnlichen Programmpunkt entspricht. Da der Compiler GAJA für jeden Programmpunkt ein Eigenschaftsobjekt (PPE) erzeugt, hat der Benutzer in diesem Fall zur Laufzeit der Animation die Wahl, ob er den Effekt eines Events in einer Sicht beobachten möchte oder nicht. IEs können in der GUI (siehe Abb. 3.2 auf S. 33) aktiviert bzw. deaktiviert werden. Deaktivierte IEs werden an die Sichten weitergeleitet. Diese müssen jedoch so programmiert sein, daß sie für deaktivierte Events nur möglicherweise notwendige, interne Änderungen am Zustand, jedoch keine graphischen Ausgaben erzeugen. Inkonsistenzen können auf diese Weise verhindert werden. Die oben beschriebene Sichtendeklaration registriert die angegebene Sicht bei der Kontrolle. Diese leitet alle zur Laufzeit auftretenden IEs zu den registrierten Sichten weiter. Ein Eventhandler einer Sicht kann aber auch selbst neue Sichten kreieren. Dazu muß er diese jeweils bei der Visualisierungskontrolle manuell registrieren.

Alternative Interesting Events

GANILA unterstützt die Gruppierung von Programmpunkten durch deren Einklammerung mit ***{** und ***}**. Das Statement ***FOLD *{** *<Eventliste>* ***}** löst ein oder mehrere alternative IEs anstelle der IEs aus, die möglicherweise in der geklammerten Programmpunktgruppe enthalten sind. Das folgende Programmbeispiel zeigt die Anwendung dieses Konstrukts:

```
        *{ help = A[i];   *IE_MoveToTemporary(A, i);
           A[i] = A[j];   *IE_MoveElement(A, i, j);
           A[j] = help;   *IE_MoveFromTemporary(A, j);
        *}
*FOLD *{ *IE_Exchange(A, i, j); *}
```

Unter Verwendung der GUI kann ein Benutzer zur Laufzeit entscheiden, ob entweder die drei IEs vor dem Auftreten des Schlüsselworts ***FOLD** ausgeführt werden oder stattdessen das einzelne Event ***IE_Exchange(A, i, j)** abgearbeitet wird. In beiden Fällen aber wird der Programmcode in der ersten Klammerung abgearbeitet, damit der Berechnungszustand des Programms bis auf die Eventausführung gleich bleibt. In einer alternativen Animation unseres Beispiels könnten sich die beiden Werte des Feldes gleichzeitig zu ihren neuen Positionen innerhalb des Feldes A bewegen. Zu beachten ist aber, daß es sich hierbei um eine nebenläufige Implementierung des entsprechenden Eventhandlers einer Sicht handelt und nicht um eine Anwendung des unten beschriebenen Paralleloperators ***II**. ***FOLD**-Konstrukte ermöglichen in GANILA das sogenannte *Semantische*

Zoomen, d. h. im Regelfall produzieren die im vorangehenden Block befindlichen Events feinkörnigere Animationen als die alternativen Events, die dem Schlüsselwort ***FOLD** folgen.

Alternative Blöcke

Im Gegensatz zum ***FOLD**-Operator erlaubt das GANILA-Konstrukt ***ALT** die Angabe zweier alternativer Programmblöcke, die das gleiche Ergebnis produzieren sollten. Der Betrachter einer Animation kann dann zur Laufzeit entscheiden, welcher Block gegenwärtig auszuführen ist.

```
      *{ min = A[0];
         for (int i = 1 ; i < A.length ; i++) {
            *IE_Compare(A, i, min);
            if (A[i] < min) min = A[i];
         }
      *}
*ALT *{ min = A[A.length];
         for (int i = A.length ; i >= 0 ; i--) {
            *IE_Compare(A, i, min);
            if (A[i] < min) min = A[i];
         }
      *}
```

Dies ist insbesondere dann nützlich, wenn unterschiedliche Berechnungswege dargestellt werden sollen. Das Programmbeispiel bestimmt den kleinsten Wert eines Feldes A. Im ersten Fall testet die Schleife jeweils zwei Feldelemente im Feldindex von unten nach oben, im zweiten Fall von oben nach unten ab. Das identische Interesting Event ***IE**_Compare(A, i, min) visualisiert den Vergleich in beiden Fällen.

Parallele Ausführung

Einzelne Programmpunkte bzw. Gruppen von Programmpunkten können mit Hilfe des Operators ***II** parallel ausgeführt werden.

```
   *{ help1 = A[i]; *IE_AssignTemporary(A, 1, i); *}
*II *{ help2 = A[j]; *IE_AssignTemporary(A, 2, j); *}
   *{ A[j] = help1; *IE_MoveTemporary(A, j, 1); *}
*II *{ A[i] = help2; *IE_MoveTemporary(A, i, 2); *}
```

Im oben dargestellten Programm werden zuerst die Zuweisungen zu help1 und help2 parallel ausgeführt und anschließend die Zuweisungen zu A[i] und A[j], die jeweiligen Interesting Events eingeschlossen. Als Resultat werden die korrespondierenden Animationen parallel dargestellt. Zu beachten ist, daß Datenabhängigkeiten eine parallele Ausführung nicht erlauben würden, falls nur *eine* Hilfsvariable help zum Vertauschen der Variablenwerte verwendet worden wäre. Im obigen Fall mußte der Algorithmus daher leicht

umgeschrieben werden, um die gewünschte Animation zu ermöglichen. Der Operator ***II** erzeugt, startet und synchronisiert für jeden der beiden Blöcke intern zwei Java-Threads. Auch dieses Konstrukt stellt einen eigenen Programmpunkt dar und erlaubt es, zur Laufzeit von paralleler auf sequentielle Ausführung einer Animation umzuschalten und umgekehrt.

Test von Invarianten

Nicht selten ist man an Eigenschaften von Algorithmen interessiert, die für alle Programmzustände oder eine Teilmenge davon erfüllt sein sollen. Der Entwickler einer Animation hat im Ganimal-Framework die Möglichkeit, eine Hypothese über einen Booleschen Ausdruck vorzugeben und diese an einer Menge von Programmpunkten bzgl. ihrer Gültigkeit dynamisch zu testen. Im nachfolgenden Programmbeispiel wird die sogenannte Heapeigenschaft an einem Teilcode des Heapsort-Algorithmus überprüft, der die Werte eines Feldes A[] sortiert.

```
*IV(A[i] >= A[2 * i + 1] && A[i] >= A[2 * i + 2])
*{
  // Teil des Programmcodes eines Heapsort-Algorithmus
*}
```

Handelt es sich bei der über das Konstrukt ***IV**(<*CondExpr*>) angegebenen Hypothese um eine Invariante für die nachfolgende Programmpunktgruppe, so erkennt der Benutzer über eine Invariantensicht zur Laufzeit der Animation, an welchen Programmpunkten die Invariante verletzt und ab welchem Punkt sie wiederhergestellt wird (vgl. Abschnitt 5.4.2 auf S. 95). Wenn der angegebene Boolesche Ausdruck eine Invariante der Programmpunktgruppe ist, dann sollte diese niemals zu **false** ausgewertet werden, wenn das System die einzelnen Programmpunkte später nacheinander abarbeitet. Das Laufzeitsystem wertet auch ineinander verschachtelte Hypothesen korrekt aus. Dabei kann der Anwender ebenfalls einzelne mit ***IV** übergebene Hypothesen über die GUI deaktivieren, weil es sich auch hier um einen eigenen Programmpunkt mit einem Eigenschaftsobjekt handelt. Manchmal ist es notwendig, daß komplexere Funktionen innerhalb des angegebenen Ausdrucks aufrufbar sind, die vom Animationsentwickler im zugrundeliegenden Ganila-Programm spezifiziert werden müssen. Da es nicht sinnvoll ist, jede Methode des Programms von außen aufrufbar zu halten, sollte der Entwickler derartige Methoden mit dem Modifikator **interactive** versehen, der auf Seite 47 näher beschrieben wird.

```
interactive boolean heapProperty(A, i) {
  // Testet Heapeigenschaft und liefert true bzw. false zurück
}
```

Die Visualisierung von Invarianten in Ganila ist ein Beispiel für zustandsgetriebene Animation, wie sie in Abschnitt 2.2.2 diskutiert wurde. Die Sicht hat hier einen direkten Zugriff auf den Programmzustand und paßt ihre Visualisierung dem permanent stattfindenden Zustandswechsel an.

Haltepunkte

Einzelne Programmpunkte können mit dem Schlüsselwort ***BREAK** als Haltepunkte markiert werden. Das Laufzeitsystem unterbricht die Ausführung des Algorithmus, wenn ein derartig gekennzeichneter Programmpunkt erreicht wird. Der Benutzer kann dann z. B. den gegenwärtigen Zustand untersuchen oder den Algorithmus schrittweise weiter ausführen. Das Konstrukt ist kein selbständiger Programmpunkt, sondern eine Eigenschaft des unmittelbar folgenden Programmpunkts.

Animationsmodi

Normalerweise geschieht die Ausführung eines Algorithmus im *PLAY*-Modus, in welchem alle auftretenden Interesting Events ohne Verzögerung ausgeführt und an jede registrierte Sicht versendet werden. Dieser Ansatz wird in der Softwarevisualisierung als *live*- oder auch *online*- Visualisierung bezeichnet. Das GANIMAL-Framework unterstützt darüber hinaus die Aufzeichnung und spätere Wiederversendung von auftretenden IEs. In diesem Fall produzieren die Eventhandler der Sichten zwar keinen graphischen Output, können aber, wenn notwendig, ihren internen Zustand an den Programmzustand angleichen. Der nachfolgend abgedruckte GANILA-Code zeigt, wie man einen Algorithmus annotiert, um eine solche *post mortem*-Visualisierung zu erhalten:

```
⋮
*RECORD;
  // Annotierter Programmcode, der aufzuzeichnende IEs enthält
*REPLAY;
⋮
```

Die Schlüsselwörter ***RECORD** und ***REPLAY** werden verwendet, um einen bestimmten Modus zur Ausführung der graphischen Primitive auszuwählen. Die Instruktion ***RECORD** wählt den *RECORD*-Modus. In diesem Modus werden zwar nicht alle IEs graphisch ausgeführt, aber durch die Visualisierungskontrolle in der Reihenfolge ihres dynamischen Auftretens abgespeichert. Die Instruktion ***REPLAY** führt zunächst alle aufgezeichneten Events graphisch aus, um anschließend wieder in den *PLAY*-Modus zurückzuschalten.

Viele naive post mortem-Visualisierungssysteme arbeiten nach diesem Prinzip: Sie spielen aufgezeichnete Events einfach wieder ab. Obwohl sie zu dem Zeitpunkt des Wiederabspielens das gesamte Wissen darüber haben, was vorher passiert ist, nutzen sie diese Tatsache oft nicht dazu aus, die visuelle Ausgabe zu verbessern. Als ein einfaches Beispiel betrachten wir eine Visualisierung, die eine textuelle Ausgabe zeilenweise in ein Fenster schreibt. Falls wir eine online-Visualisierung verwenden, müssen wir das Fenster mit einer Scrollfunktion versehen. Im Falle der post mortem-Visualisierung können wir den gesamten Text untersuchen, bevor wir auch nur eine Zeile ausgegeben haben. Mit diesem Wissen könnte man z. B. die Fenster- bzw. Schriftgröße anpassen, damit der gesamte Text in das Fenster paßt.

Oft führen auch Animationen für Algorithmen, die Graphen verändern, d. h. Knoten oder Kanten hinzufügen oder löschen, zu Konfusionen, weil nach jeder Änderung ein neues Layout des aktuellen Graphen berechnet werden muß. In diesem neuen Layout werden evtl. Knoten zu neuen Koordinaten bewegt, ohne daß der Algorithmus diese Knoten verändert hat. Daher ist es für den Anwender nicht sofort klar, welche Änderungen des Graphen durch den Algorithmus vorgenommen wurden. GANILA unterstützt durch die Animationsmodi ein sog. Vorausschauendes Graphlayout. Das heißt ein Graph wird zum Zeitpunkt t_1 gezeichnet, wobei Informationen darüber verwendet werden, wie dieser Graph zu einem späteren Zeitpunkt t_2 aussehen wird (siehe Abschnitt 5.4.3). Unabhängig von dieser Anwendung im Graphlayout ermöglichen die beiden Modi *RECORD* und *PLAY* eine Vermischung von post mortem- und live/online-Animation innerhalb derselben Algorithmenanimation, in der es beispielsweise auch möglich ist, zwischen den Aufzeichnungen Benutzereingaben zuzulassen. Diese Eigenschaft weisen die in Kapitel 2 beschriebenen Animationssysteme nicht auf.

Animationssteuerung von Schleifen und Rekursionen

Oftmals finden interessante Ereignisse innerhalb von Schleifen oder rekursiven Funktionsaufrufen statt und werden dementsprechend mit Interesting Events annotiert, man betrachte z. B. das Durchlaufen einer Liste oder rekursive Sortierverfahren wie Quicksort usw. Ist nun ein solcher Schleifendurchlauf ein Bestandteil eines größeren Algorithmus, dann ist es gegebenenfalls nicht wünschenswert, daß alle Schleifendurchläufe auch visualisiert werden. Für den Benutzer könnte es schnell langweilig werden, etwa 100 Durchläufe einer Schleife zu sehen, wenn es für das Verständnis ausreichend ist, nur die letzten fünf zu verfolgen.

Die in GANILA realisierte Lösung dieses Problems basiert auf den bereits diskutierten Animationsmodi. Allerdings mit dem Unterschied, daß nach der Aufzeichnung aller in einer Schleife oder Rekursion aufgetretenen IEs lediglich eine Teilmenge dieser Events am Ende der Schleife bzw. Rekursion wieder abgespielt wird. Die Visualisierungskontrolle gewährleistet auch hier wieder die Konsistenz der internen Zustände aller angemeldeten Sichten. Um diese selektive Visualisierung zu erreichen, erlaubt GANILA die Annotation der Schleifenkonstrukte von JAVA (**do**, **while**, **for**) mit speziellen Visualisierungsbedingungen. Diese werden in eckigen Klammern hinter die Schleifenbedingung eingefügt:

```
for(int i = 0 ; i < 100 ; i++) [$i >= $n − 5] {
  foo(i);
}
```

In der Animation dieses Programmbeispiels werden lediglich die letzten fünf Aufrufe der Funktion foo(i) visualisiert. Hier repräsentiert das Schlüsselwort $i den aktuellen Durchlauf und $n die maximale Anzahl der Durchläufe. Beide Werte stehen normalerweise erst zur Laufzeit des Programms fest und müssen jeweils neu berechnet werden. Die Visualisierungskontrolle zeichnet jedes IE solange auf, bis die letzte Iteration der Schleife erreicht ist. Dann ist auch der Wert von $n bekannt und das System kann die

für die Visualisierung relevanten IEs an die Sichten senden[1]. Beachtenswert sind hierbei mögliche Vorkommen von Schleifenaustritten mittels des JAVA-Sprachkonstrukts **break** im Rumpf der Schleife, so daß in solchen Fällen eventuell kein Durchlauf der Schleife visualisiert wird.

3.2 Reifikation von GANILA-Programmen

Die Reifikationstechnik, die hier vorgestellt wird, basiert auf Source-to-Source-Transformationen [DK02b]. Der Compiler GAJA generiert für jeden Programmpunkt eine JAVA-Methode, die den aktuellen Programmcode für den jeweiligen Programmpunkt enthält, sowie ein Objekt, welches bestimmte Metainformationen über den Programmpunkt abspeichert. Wir verwenden den Begriff *Programmpunkteinstellungen* (PPEs), um uns auf diese Metainformationen zu beziehen. Im Idealfall sollte ein reifizierter Programmpunkt sowohl die PPEs als auch den Programmcode enthalten. Eine Konsequenz dieser Vorgehensweise wäre aber, daß die Anzahl der generierten Klassen proportional zur Anzahl der Programmpunkte wäre. Die getrennte Handhabung der Programmpunkte und der PPEs ermöglicht uns, daß wir nur eine feste Anzahl von Klassen für die verschiedenen PPEs benötigen, während wir für jede Klasse des Quellprogramms exakt eine Zielklasse mit genau einer Methode pro Programmpunkt generieren.

Bevor wir unseren Ansatz im einzelnen darlegen, diskutieren wir im nächsten Abschnitt alle Metainformationen, die in unserem System über einen Programmpunkt abgespeichert werden können. Wir beschreiben in den Abschnitten 3.2.2 und 3.2.3 die Reifikation von reinen JAVA-Programmen. Anschließend erläutern wir in Abschnitt 3.2.4 die Reifikation einiger GANILA-Annotationen. Zum Abschluß führen wir eine Reihe verwandter Arbeiten zu diesem Thema auf und grenzen den hier verwendeten Reifikationsbegriff zu ähnlichen Techniken ab.

3.2.1 Programmpunkteinstellungen

Nach der Reifikation können wir mit jedem Programmpunkt zusätzliche Informationen assoziieren und diese zur Laufzeit unter Verwendung einer graphischen Benutzerschnittstelle (siehe Abb. 3.2 auf S. 33) — bzw. wann immer der Programmpunkt ausgeführt wird — ändern. In der gegenwärtigen Implementierung des GANIMAL-Frameworks sind die folgenden PPEs fest eingebaut:

- Informationen, die mit jedem Programmpunkt verknüpfbar sind:
  - *Haltepunkte.* Der aktuelle Programmpunkt ist ein Haltepunkt, der in GANILA mit einem voranstehenden ***BREAK** markiert wurde. Die Ausführung wird gestoppt, und der Benutzer kann den aktuellen Zustand untersuchen oder die Ausführung fortsetzen.

[1] Analog ist dieses Vorgehen auch für rekursive Funktionsaufrufe vorgesehen, wobei hier `$i` die aktuelle Rekursionstiefe und `$n` die maximale Tiefe beschreibt. In der gegenwärtigen Version des Frameworks ist dieses Merkmal noch nicht vollständig implementiert und getestet.

  - *Modus.* Der aktuelle Programmpunkt wurde mit einem der beiden Schlüsselwörter ***RECORD** oder ***REPLAY** gekennzeichnet. An diesem Programmpunkt wird folglich einer der beiden Animationsmodi *RECORD* oder *PLAY* gesetzt.

- Informationen, die nur mit Schleifenkonstrukten und Methodendeklarationen verknüpfbar sind:

  - *#Besuche.* Bei jeder Ausführung des aktuellen Programmpunkts für eine **for**-, **while**- oder **do**-Schleife bzw. für eine Methodendeklaration kann ein Zähler initialisiert oder erhöht werden, um die Anzahl der Iterationen einer Schleife bzw. die Tiefe eines rekursiven Aufrufs zu zählen.

- Informationen, die nur mit GANILA-Konstrukten verknüpfbar sind:

  - *Interesting Events.* Der aktuelle Programmpunkt ist ein Interesting Event. Dieses kann aktiv sein oder nicht, d. h. die mit diesem Event verbundene graphische Visualisierung wird ausgeführt oder nicht.
  - *Invarianten.* Dieser Programmpunkt umschließt einen Block, für den eine bestimmte Invariante getestet werden soll. Die Invariante erscheint beim Eintritt in den Block in der Invariantensicht und wird beim Verlassen wieder entfernt. Auch hier kann die Überprüfung der Invariante ein- oder ausgeschaltet sein.
  - *Parallele Ausführung.* Dieses ist der Programmpunkt eines Paralleloperators ***||**. Seine PPE bestimmt, ob seine beiden Blöcke sequentiell oder parallel ausgeführt werden.
  - *Alternativen.* Dieses ist der Programmpunkt eines ***ALT**-Operators. Seine PPE bestimmt, welcher seiner beiden Blöcke ausgeführt wird.
  - *Falten.* Dieses ist der Programmpunkt eines Operators ***FOLD** für alternative IEs, der aus einem oder mehreren IEs und einem Block besteht. Wenn sein Syntaxbaumknoten in der GUI aktiviert ist, dann werden die alternativen IEs gesendet, aber keines der IEs, die möglicherweise im Block enthalten sind. Ist er inaktiv, so werden die IEs im Block gesendet. In beiden Fällen werden die Instruktionen im Block ausgeführt.

Man könnte in den PPEs auch eine Referenz zum abstrakten Syntaxbaum des Programmpunkts abspeichern. Dies wurde bisher von unseren Anwendungen noch nicht vorausgesetzt.

3.2.2 Anweisungen

Wir werden nun einige Beispielprogramme untersuchen, um zu illustrieren, wie der Compiler GAJA Programmpunkte reifiziert. Zunächst diskutieren wir die Übersetzung reiner JAVA-Programme unter Betrachtung von Instruktionssequenzen, Methodenaufrufen und Kontrollstrukturen. Im darauffolgenden Abschnitt werden wir dann auf die Übersetzung

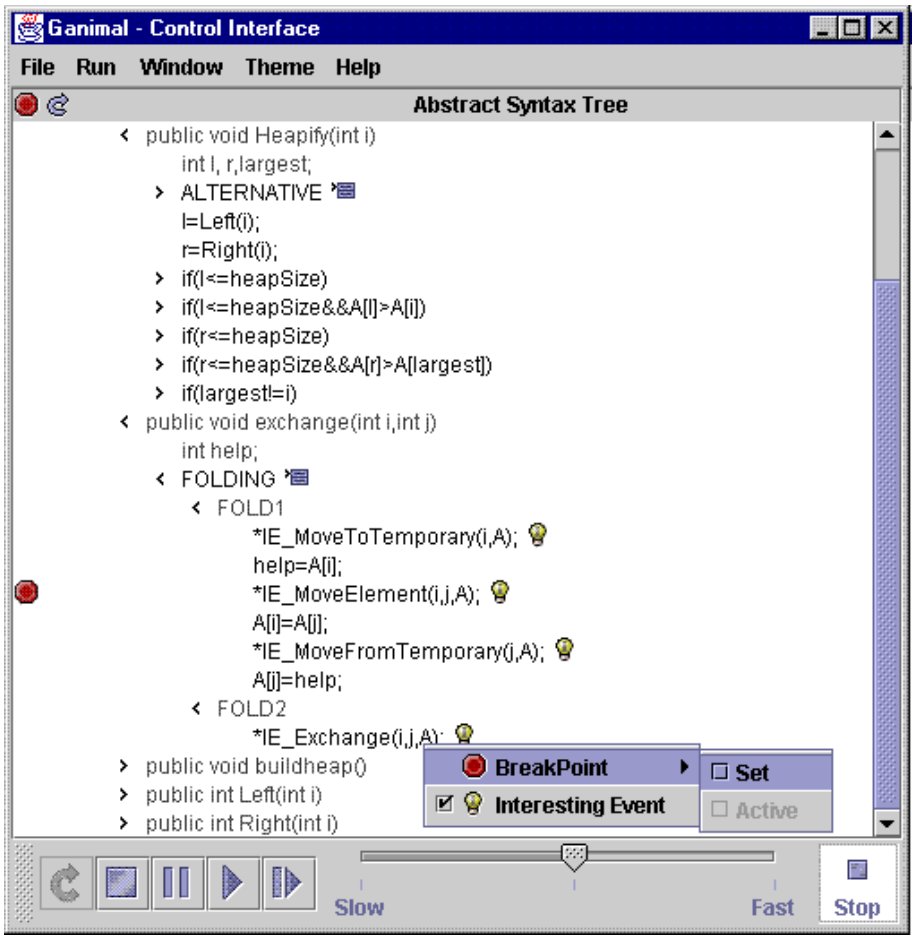


Abbildung 3.2: Graphische Benutzerschnittstelle zur Modifikation der PPEs.

der durch GANILA bereitgestellten Spracherweiterungen näher eingehen. Um die Lesbarkeit der Beispielprogramme zu erhöhen, wird auf den Code für die Ausnahmebehandlung verzichtet und die Abkürzung PP anstelle des Wortes „Programmpunkt“ verwendet.

Naiver Ansatz

In unserem ersten Beispiel definieren wir eine Klasse Example1 mit einer einzigen Methode start(). Dabei wird auf eine Objektvariable x und eine lokale Variable y im Methodenrumpf zugegriffen.

```
ganila class Example1 {
  int x;

  public void start(String[] argv) {
    int y = 2;
    x = x + y;                                          // PP 0
```

```
    y = x * y;                                                    // PP 1
  }
}
```

Aus dem Quellcode der Klasse Example1 generiert GaJa eine neue Java-Klasse Example1Algorithm. Um die beiden Programmpunkte in dieser Methode zu reifizieren, generieren wir jeweils eine Methode dispatch_0() für den ersten und eine Methode dispatch_1() für den zweiten Programmpunkt:

```
public class Example1Algorithm {
  Gint x = new Gint();
  public void start(String[] argv) {
    Gint y = new Gint(2);
    passControlToGUI(0); dispatch_0(y);
    passControlToGUI(1); dispatch_1(y);
  }
  public void dispatch_0(Gint a1) {
    x.setValue(x.getValue() + a1.getValue());
  }
  public void dispatch_1(Gint a1) {
    a1.setValue(x.getValue() * a1.getValue());
  }
}
```

Beide Variablen x und y müssen von den generierten Methodenrümpfen aus zugreifbar sein. Da die generierten Methoden als Objektmethoden innerhalb derselben Klasse definiert sind, ist die Klassenvariable x noch von diesen Methoden aus zugreifbar. Dagegen muß die lokale Variable y den generierten Methoden explizit als Argument übergeben werden. Um Variablen von einem Kontext zum anderen zu übergeben, stellen wir die Wrapperklassen Gint, Gboolean etc. für alle primitiven Typen zur Verfügung[2]. Wir verwenden verpackte Werte als L-Werte und unverpackte Werte als R-Werte.

Durch den Aufruf der Methode passControlToGUI() vor der Ausführung eines jeden Programmpunkts erhält die GUI Zugriff auf den aktuellen Zustand und kann somit Informationen abrufen. Zum Beispiel könnte sie überprüfen, ob der aktuelle Programmpunkt ein Haltepunkt ist, oder auf eine Benutzereingabe warten, um die Ausführung fortzusetzen, oder einfach einige Millisekunden warten, um die Ausführung etwas zu verzögern. Weiterhin könnte sie testen, ob eine Invariante immer noch gültig ist, und sie könnte den aktuellen Programmpunkt in der CodeView, einem Fenster, das den Quellcode enthält, markieren.

[2]Die Wrapperklassen Integer, Boolean etc. im Standardpaket java.lang sind final [Fla00], daher mußten eigene Wrapperklassen definiert werden. Diese ermöglichen auch, Visualisierungsmethoden und zusätzliche Informationen hinzuzufügen.

Fortgeschrittener Ansatz

Um die Wartbarkeit des Compilers zu verbessern, erbt nun die neu generierte Klasse Example1Algorithm die vordefinierte generische Methode dispatch() von einer Klasse des Laufzeitsystems GAlgorithm, die wiederum die Methode passControlToGUI() aufruft. Um einen Programmpunkt auszuführen, wird ein Aufruf einer allgemeinen Methode dispatch() anstelle des ursprünglichen Programmpunkts generiert:

```
public class Example1Algorithm extends GAlgorithm {
  ...
  public void start(String[] argv) {
    Gint y = new Gint(2);
    dispatch(0, new Object[]{ y });
    dispatch(1, new Object[]{ y });
  }
  ...
}
```

Die Methode dispatch(pp, **new** Object[]{ $arg_1, \ldots, arg_n$ }) ruft für den Programmpunkt pp ihrerseits die Methode dispatch_pp($arg_1, \ldots, arg_n$) via Reflexion (siehe Abschnitt 3.2.5) auf. Wenn wir nun das Aufrufverhalten zu einem späteren Zeitpunkt ändern wollen, müssen wir nicht den Compiler ändern. Es reicht aus, die Klassendefinition von GAlgorithm zu modifizieren. Dieser Ansatz implementiert das Entwurfsmuster *Strategy* [GHJV95] und erleichert damit zusätzlich die Algorithmenauswahl über die graphische Benutzerschnittstelle. Die Klasse GAlgorithm ist in folgendem Codefragment teilweise wiedergegeben:

```
public class GAlgorithm {
  ...
  public synchronized ControlFlow dispatch(int pp, Object[] args) {
    passControlToGUI(pp);
    Class c = getClass(), params[];
    if (args == null) params = new Class[0];
    else params = new Class[args.length];
    params = new Class[];
    for (int i = 0; i < params.length; i++) {
      params[i] = args[i].getClass();
    }
    Method m = c.getMethod("dispatch_" + pp, params);
    Object o = m.invoke(this, args);
    if (o instanceof ControlFlow) return (ControlFlow) o;
    return new ControlFlow();
  }
}
```

3.2.3 Kontrollstrukturen

Im obigen Beispiel Example1 hatten wir einen sehr einfachen Kontrollfluß: lediglich eine Sequenz von zwei Zuweisungen. Im folgenden Beispiel Example2 mit vier Programmpunkten untersuchen wir einen Methodenaufruf mit Rückgabewerten sowie eine bedingte Anweisung. Weitere Kontrollstrukturen wie Schleifen oder die **switch**-Anweisung werden in analoger Weise übersetzt.

```
ganila class Example2 {
  int x;

  public void start(String[] argv) {
    int y = 2;
    x = max(x, y);                                          // PP 0
  }
  public int max(int a, int b) {
    if(a > b)                                               // PP 1
      return a;                                             // PP 2
    else
      return b;                                             // PP 3
  }
}
```

Wieder generieren wir Methoden für jeden der vier Programmpunkte. Dabei führen wir ControlFlow-Objekte ein, um die Verzweigungsanweisungen **break**, **continue** und **return** zu behandeln[3]. Durch die Verwendung unserer Reifikationstechnik werden eingebettete Blöcke zu eingebetteten Aufrufen von dispatch()-Methoden übersetzt. Die Ausführung von einer der oben aufgeführten Anweisungen wird die Abarbeitung des aktuellen Blocks beenden und in einem der umschließenden Blöcke fortfahren. Im folgenden werden wir eine Referenz zu diesem umschließenden Block als *Zielblock* bezeichnen. Im übersetzten Programm wird die Terminierung eines Blocks dadurch implementiert, daß die dispatch()-Methode mit einem ControlFlow-Objekt als Rückgabewert verlassen wird. Alle umschließenden dispatch()-Methoden müssen diesen Wert an ihre jeweiligen Aufrufer weiterleiten, bis die Übersetzung des Zielblocks erreicht wird. In unserem Programmbeispiel entspricht der Zielblock der **return**-Anweisungen dem Rumpf der Methode max(). Wenn das ControlFlow-Objekt selbst eine **return**-Anweisung repräsentiert, dann wird die Anweisung an dieser Stelle mit dem im ControlFlow-Objekt enthaltenen Wert als Rückgabewert ausgeführt:

```
public class Example2Algorithm extends GAlgorithm {
  Gint x = new Gint();

  public void start(String[] argv) {
    Gint y = new Gint(2);
```

[3] In der Sprachspezifikation von JAVA [GJSB00] werden diese drei Anweisungen explizit als *Verzweigungsanweisungen* (Branching Statements) bezeichnet.

```
    dispatch(0, new Object[]{ y });
  }
  public Gint max(Gint a, Gint b) {
    ControlFlow cfl;
    cfl = dispatch(1, new Object[]{ a, b });
    return (Gint) cfl.value;
  }
  public void dispatch_0(Gint a1) {
    x.setValue(max(x.Clone(), a1.Clone()).getValue());
  }
  public ControlFlow dispatch_1(Gint a1, Gint a2) {
    ControlFlow cfl;
    if(a1.getValue() > a2.getValue()) {
      cfl = dispatch(2, new Object[]{ a1 });
      if (cfl.isReturn) return cfl;
    } else {
      cfl = dispatch(3, new Object[]{ a2 });
      if (cfl.isReturn) return cfl;
    }
    return new ControlFlow();
  }
  public ControlFlow dispatch_2(Gint a1) {
    return new ControlFlow("return", new Gint(a1.getValue()));
  }
  public ControlFlow dispatch_3(Gint a1) {
    return new ControlFlow("return", new Gint(a1.getValue()));
  }
}
```

Das **throw**-Statement von JAVA muß allerdings nicht in einem ControlFlow-Objekt verpackt werden, da **throw** automatisch alle eingeschlossenen Blöcke oder Methodenaufrufe terminiert, bis es seine entsprechende **catch**-Klausel erreicht.

3.2.4 Ausgewählte GANILA-Konstrukte

Bisher wurde beschrieben, wie GAJA einfache JAVA-Anweisungen und -Kontrollstrukturen für deren Reifikation übersetzt. Diese Programmpunkte erhalten in unserer Implementierung bis auf *Haltepunkt* und *Modus* keine PPEs (siehe Abschnitt 3.2.1). Im folgenden präsentieren wir die Übersetzung einer Auswahl von GANILA-Annotationen, deren Repräsentationen zur Laufzeit mit speziellen PPEs versehen werden. Der für die verbleibenden Annotationen zu generierende Code wird in analoger Weise gebildet.

Interesting Events

Als erste mögliche Annotation fügten wir in Abschnitt 3.1.1 Interesting Events als Methodenaufrufe mit dem Präfix ***IE_** hinzu. Diese transferieren lokale Informationen in ihren Argumenten zu den verschiedenen Sichten. Betrachten wir das folgende Codebeispiel, das aus einer Methodendeklaration isPrime() und einem Interesting Event ***IE_**Prime(n) (z. B. an Programmpunkt mit der Nummer 4) innerhalb des Methodenrumpfes besteht:

```
public boolean isPrime(int n) {
  *IE_Prime(n);                                          // PP 4
  // Rest des Methodenrumpfes
  ...
}
```

Zur Laufzeit des Algorithmus wird ein Eventobjekt der Klasse GEvent anstelle des o. g. Methodenaufrufs kreiert. Dieses Objekt enthält folgende Informationen: den aktuellen Programmpunkt, den aktuellen Animationsmodus als Ergebnis des Prädikats isRecord(), den Aktivierungszustand (PPE) des Events durch isVisibleEvent(), die Umgebung der Metavariablen $i und $n in der kontrollierten Visualisierung von Schleifen und Rekursion getLoReDepth(), den Namen des Events und die Argumente des ursprünglichen Methodenaufrufs.

```
public Gboolean isPrime(Gint n) {
  dispatch(4, new Object[]{ n });
  // Übersetzter Code für den Rest des Methodenrumpfes
  ...
}
public void dispatch_4(Gint a1) {
  GEvent e;
  e = new GEvent(4,
        isRecord(),
        isVisibleEvent(),
        getLoReDepth(),
        "Prime", new Object[]{ a1 });
  control.broadcast(e);
}
```

Wie im Beispiel zu sehen ist, wird das GEvent-Objekt durch ein Kontrollobjekt der Visualisierungskontrolle versendet (vgl. Abschnitt 5.3). Dieses leitet das GEvent-Objekt über einen Broadcast zu allen angemeldeten Sichten weiter. Jede Sicht kann individuell auf die PPEs reagieren und entscheiden, ob sie den eigenen Eventhandler für diese IEs aufruft oder nicht. Bis hierhin differiert unsere Arbeitsweise der IEs nicht sehr von der in anderen event-basierten Algorithmenanimationssystemen mit Mehrfachsichten wie z. B. Zeus [Bro91]. Der Unterschied besteht darin, daß zusätzliche Informationen in den

Eventobjekten gespeichert sind, und daß unsere Programmpunkte reifiziert werden. Deshalb bietet unser System einige Funktionalitäten, die in anderen bekannten Algorithmenanimationssystemen [EZ96, SDBP98, DK00b, Die02] nicht enthalten sind. Der Gebrauch und die Implementierung dieser Eigenschaften werden im folgenden beschrieben.

Animationsmodi

Natürlich kostet die Aufzeichnung und Wiedergabe von Events durch den *RECORD*- und *PLAY*-Mechanismus einiges an zusätzlichem Aufwand, um es den Sichten im Falle einer post mortem-Visualisierung zu ermöglichen, eine bessere graphische Ausgabe zu produzieren. Im GANIMAL-Framework muß eine Sicht für jedes Interesting Event ***IE_**<*Eventname*> die vier nachfolgenden Methoden implementieren:

- IE_*Eventname*_Play_Visible() produziert eine visuelle Ausgabe sowie interne Zustandsänderungen.
- IE_*Eventname*_Play_Invisible() produziert keine visuelle Ausgabe, aber interne Zustandsänderungen. Diese Methode wird für deaktivierte IEs aufgerufen, so daß später aktivierte Events auf einem konsistenten internen Zustand ausgeführt werden können.
- IE_*Eventname*_Record_Visible() produziert weder visuelle Ausgaben, noch interne Zustandsänderungen, sondern berechnet nur Informationen, die in einer post mortem-Visualisierung verwendet werden. Wenn das Event später wieder auf der Methode IE_*Eventname*_Play_Visible() abgespielt wird, kann es diese zusätzliche Information ausnutzen.
- IE_*Eventname*_Record_Invisible() produziert weder visuelle Ausgaben, noch interne Zustandsänderungen, sondern berechnet nur Informationen, die in einer post mortem-Visualisierung verwendet werden. Wenn das Event später wieder auf der Methode IE_*Eventname*_Play_Invisible() abgespielt wird, kann es diese zusätzliche Information ausnutzen.

Diese Vorgehensweise sieht nach sehr viel Arbeit für den Visualisierer aus, aber der Compiler GAJA produziert für die Sichten sog. Templateklassen [GHJV95]. Wenn der Programmierer nur online-Visualisierung und naive post mortem-Visualisierung möchte, dann muß er lediglich den Code für die Methode IE_*Eventname*_Play_Visible() schreiben. Um die erweiterte post mortem-Visualisierung zu nutzen, müssen die verbleibenden drei Methoden ausprogrammiert werden.

Animationssteuerung von Schleifen

In diesem Abschnitt erläutern wir, auf welche Art und Weise das Zählen der Besuche von Programmpunkten und das Wechseln zwischen verschiedenen Animationsmodi dazu verwendet werden kann, ausgewählte Iterationen einer Schleife zu visualisieren. In der Sprache GANILA werden Visualisierungsbedingungen in eckigen Klammern hinter die Schleifenbedingung geschrieben, wie Beispiel Example3 verdeutlicht:

```
ganila class Example3 {
  public void start(String[] argv) {
    int s = 0;
    for(int j = 0; j < 100; j++) [($n − $i < 5) && ($i % 2 == 1)] {        // PP 0
      if(isPrime(j)) {                                                    // PP 1
        s = s + j;                                                        // PP 2
        *IE_Add(s, j);                                                    // PP 3
      }
    }
  }
  public boolean isPrime(int n) {
    *IE_Prime(n);                                                         // PP 4
    ...
  }
}
```

Hier beschreibt die Variable $i die Nummer der aktuellen Iteration und die Variable $n die maximale Anzahl von Iterationen der jeweiligen Schleife. Wichtig ist, daß beide Werte nur zur Laufzeit des Algorithmus berechnet werden können. Im Beispiel werden nur diejenigen der letzten fünf Iterationen visualisiert, die ungerade numeriert sind, d. h. die Events IE_Prime(95), IE_Prime(97), IE_Add(963, 97) und IE_Prime(99).

Die GANILA-Klasse Example3 wird analog wie bisher übersetzt, allerdings generieren wir auch spezielle PPEs (Settings) für den Programmpunkt 0, d. h. für den Programmpunkt der Schleife und erweitern die generische dispatch()-Methode. Diese speichert alle Events, bis die letzte Iteration erreicht wird. Dann ist der Wert von $n bekannt, und die generische dispatch()-Methode kann alle relevanten Events wieder abspielen. Die Visualisierungsbedingung als solche wird von GAJA in eine Testmethode checkVisits() transformiert, die als ein Teil der Einstellungen für den Programmpunkt (hier 0) implementiert wurde:

```
st[0] = new Settings(0, new Object[] {
  new Visits_Setting(true, "($n − $i < 5) && ($i % 2 == 1)") {
    public boolean checkVisits(int $i, int $n) {
      return ($n − $i < 5) && ($i % 2 == 1);
    }
  }
});
```

Ein Objekt für die Einstellung *#Besuche* besteht also aus einem Booleschen Wert für den Aktivierungszustand, einer Stringdarstellung für die Darstellung der Bedingung in der GUI sowie aus einer Testmethode, welche die Visualisierungsbedingung zur Laufzeit der Animation auswertet. Wir fügen weiterhin drei neue Methodenaufrufe in die allgemeine dispatch()-Methode aus Abschnitt 3.2.2 ein:

```
public synchronized ControlFlow dispatch(int pp, Object[] args) {
  passControlToGUI(pp);
  loopIncrement(pp);
    // Berechne c und params[] wie zuvor
  loopBegin(pp);
    Method m = c.getMethod("dispatch_" + pp, params);
    Object o = m.invoke(this, args);
  loopEnd(pp);
    // Liefere ControlFlow Objekt zurück wie zuvor
}
```

Bevor wir diese Funktionen erläutern, wenden wir uns zunächst einem etwas komplexeren Beispiel für die Animationssteuerung einer **while**-Schleife zu, die sich im Rumpf einer ebenfalls kontrolliert visualisierten **for**-Schleife befindet:

```
void foo() {
  for(...) [$n − $i < 5] {                                    // PP 0
    while(...) [$n − $i < 3] {                                // PP 1
      x = x + 5;                                              // PP 2
      foo();                                                  // PP 3
    }
  }
}
```

Betrachten wir nun genau die Situation, die eintritt, wenn die Methode foo() zweimal rekursiv aufgerufen wurde und der aktuelle Programmpunkt $pp = 2$ ist. In diesem Moment existieren vier lebende Instanzen der Metavariablen $i und $n. In unserer Implementierung speichern wir diese Variablen auf einem Keller: die Umgebung für die Metavariablen. Um den Wert von $n zu berechnen, ist es notwendig, alle Events aufzuzeichnen, um dann den Wert von $n nach der letzten Iteration zu erhalten und anschließend alle aufgezeichneten Events abzuspielen. Die Methoden loopIncrement() usw. manipulieren die Umgebung wie folgt:

- loopIncrement(*pp*): Schleifenzähler werden jeweils am ersten Programmpunkt innerhalb der Schleife inkrementiert. Das heißt wenn der Programmpunkt $pp - 1$ eine Schleife mit einer Visualisierungsbedingung und die Umgebung nicht leer ist, dann erhöht diese Methode die oberste Instanz von $i auf dem Keller.

- loopBegin(*pp*): Wenn der aktuelle Programmpunkt *pp* keine Schleife mit einer Visualisierungsbedingung ist, berechnet diese Methode nichts. Wenn dies doch der Fall ist, dann wird ein neuer Eintrag für die Schleifenzähler auf dem Keller erzeugt. Der Eintrag wird zum Speichern der Werte für *pp*, $i und $n genutzt. Wenn der Keller vorher leer war, dann ist gegenwärtig keine andere Visualisierungsbedingung aktiv und die Methode ändert den Animationsmodus derart, daß die aktuelle und alle umschlossenen Schleifen im *RECORD*-Modus ausgeführt werden.

- loopEnd(*pp*): Wenn der aktuelle Programmpunkt *pp* keine Schleife mit einer Visualisierungsbedingung ist, berechnet diese Methode nichts. Andernfalls entfernt die Methode den obersten Eintrag des Kellers. Wie in Abschnitt 3.2.4 gezeigt, enthält jedes IE eine Kopie der aktuellen Umgebung für Metavariablen, wobei alle Werte der Variableninstanzen von $i und alle Referenzen der Variableninstanzen von $n kopiert werden. Damit kann die Methode nun den Wert von $n auf den Wert von $i setzen, d. h. auf den Wert der letzten Iteration. Aufgrund der Referenzen ändern sich die Werte aller Kopien von $n automatisch. Wenn der Keller nun leer ist, dann läßt diese Methode alle aufgezeichneten Events durch die Visualisierungskontrolle abspielen. Anschließend wird der Animationsmodus auf seine Standardeinstellung *PLAY* zurückgesetzt.

Als Ergebnis dieser Vorgehensweise beinhaltet ein aufgezeichnetes Event beim Wiederabspielen durch die Visualisierungskontrolle eine Kopie der Umgebung mit den korrekten Werten für $i und $n, d. h. die Kontrolle ist jetzt in der Lage, alle Visualisierungsbedingungen abzutesten. Genauer betrachtet wird das Event genau dann visuell durch einen Aufruf von IE_*Eventname*_Play_Visible() ausgeführt, wenn die Testmethode st[*pp*].checkVisits(i, n) für jeden Eintrag (pp, i, n) in dieser Umgebung den Booleschen Wert **true** zurückliefert. Anderenfalls wird die Methode IE_*Eventname*_Play_Invisible() aufgerufen.

Die Animationssteuerung der Rekursion wird in analoger Weise implementiert. In diesem Fall entspricht $n der maximalen Rekursionstiefe, d. h. der maximalen Pfadlänge im dynamischen Aufrufgraphen.

Parallele Ausführung

Im vierten Beispiel Example4 dieses Abschnitts diskutieren wir die Übersetzung des Parallelоperators ***||**. Dieser ermöglicht die parallele Ausführung zweier Blöcke:

```
ganila class Example4 {
  int x = 1;
  public void start(String[] argv) {
    int y = 2, tmp1, tmp2;
    tmp1 = x;                                          // PP 0
    tmp2 = y;                                          // PP 1
      *{ *IE_MoveTo("x", "y");                         // PP 3
         x = tmp2; *}                                  // PP 4
    *||                                                // PP 2
      *{ *IE_MoveTo("y", "x");                         // PP 5
         y = tmp1; *}                                  // PP 6
  }
}
```

Im obigen Programm werden zuerst die beiden Zuweisungen auf die Hilfsvariablen tmp1 und tmp2 sequentiell nacheinander (PPs 0/1) und anschließend die beiden Zu-

weisungen auf x und y zusammen mit den jeweiligen Interesting Events parallel (PPs 3/4 parallel zu PPs 5/6) ausgeführt. Als Resultat laufen auch die korrespondierenden Animationen parallel ab. Wie auf Seite 28 bereits angedeutet, verwenden wir in diesem Beispiel zwei Hilfsvariablen. Wenn man nur eine Hilfsvariable zur Verfügung stellen würde, wäre eine parallele Ausführung wegen Datenabhängigkeiten unmöglich. Hier muß also der Algorithmus leicht verändert werden, um parallele Animationen zu erlauben.

Für die Übersetzung nach JAVA betrachten wir den Paralleloperator als eigenen Programmpunkt und übersetzen ihn in einen Aufruf der Methode executeParallel(), die von der Klasse GAlgorithm geerbt wird. In unserem Beispiel hat der Programmpunkt des Paralleloperators die Nummer 2. Für die Repräsentation seiner beiden Blöcke generieren wir die zwei Methoden execute_2_1() und execute_2_2(). Zur Vereinfachung verzichten wir auf die Übersetzung der im Programm vorkommenden IEs an den Programmpunkten 3 und 5.

```
public class Example4Algorithm extends GAlgorithm {
  Gint x = new Gint(1);
  public void start(String[] argv) {
    Gint y = new Gint(2), tmp1 = new Gint(), tmp2 = new Gint();
    dispatch(0, new Object[]{ tmp1 });
    dispatch(1, new Object[]{ tmp2, y });
    dispatch(2, new Object[]{ tmp1, tmp2, y });
  }
  public void dispatch_0(Gint a1) {
    a1.setValue(x.getValue());
  }
  public void dispatch_1(Gint a1, Gint a2) {
    a1.setValue(a2.getValue());
  }
  public ControlFlow dispatch_2(Gint a1, Gint a2, Gint a3) {
    return executeParallel(2, new Object[]{ a1, a2, a3 });
  }
  public void dispatch_4(Gint a1) {
    x.setValue(a1.getValue());
  }
  public void dispatch_6(Gint a1, Gint a2) {
    a2.setValue(a1.getValue());
  }
  public ControlFlow execute_2_1(Gint a1, Gint a2, Gint a3) {
    dispatch(4, new Object[]{ a2 });
    return new ControlFlow();
  }
  public ControlFlow execute_2_2(Gint a1, Gint a2, Gint a3) {
    dispatch(6, new Object[]{ a1, a3 });
```

```
    return new ControlFlow();
  }
}
```

Basierend auf den PPEs des Programmpunkts 2 des Paralleloperators ***II** kreiert executeParallel() zwei Threads, um die beiden generierten Methoden parallel aufzurufen, oder sie ruft diese Methoden sequentiell auf, falls der Benutzer die Parallelausführung über die GUI deaktiviert hat:

```
public class GAlgorithm {
  ...
  public synchronized ControlFlow executeParallel(int pp, Object[] args) {
    ...
    if(st[pp].isParallelActive()) {
      // Führe beide Blöcke parallel aus
      Thread t1 = new Thread(new GAlgThread(this, pp, 1, args));
      Thread t2 = new Thread(new GAlgThread(this, pp, 2, args));
      t1.start(); t2.start();
      t1.join(); t2.join();
      return chooseCFL(t1.getCFL(), t2.getCFL());
    } else {
      // Führe beide Blöcke nacheinander aus und
      // berechne die Variable c und das Feld params[] wie zuvor
      Method m = c.getMethod("execute_" + pp + "_1", params);
      Object o1 = m.invoke(this, args);
      if((o1 instanceof ControlFlow) && !((ControlFlow) o1).isEmpty())
        return (ControlFlow) o1;
      m = c.getMethod("execute_" + pp + "_2", params);
      Object o2 = m.invoke(this, args);
      if(o2 instanceof ControlFlow)
        return (ControlFlow) o2;
    }
  }
}
```

Im Fall der sequentiellen Ausführung wird das durch den Code des ersten Blocks zurückgelieferte Kontrollflußobjekt zuerst überprüft, bevor der Code für den zweiten Block ausgeführt wird. Wenn dieses beispielsweise einer **break**- oder **return**-Anweisung entspricht, darf der zweite Block nicht mehr ausgeführt werden. Im Fall paralleler Ausführung liefert eine Hilfsmethode chooseCFL() ein neues Kontrollflußobjekt zurück, das aus den entsprechenden Objekten beider paralleler Kontrollflüsse gebildet wird. Unvereinbare Zustände führen zu einem Laufzeitfehler. Allgemein gilt, daß der Programmierer besonderes Augenmerk auf solche Problematiken zu richten hat, da GAJA bisher keine statische Analyse dazu durchführt.

In ähnlicher Weise werden alternative Blöcke durch den ***ALT**-Operator sowie das Falten von Interesting Events durch den ***FOLD**-Operator in Aufrufe der generischen Methoden executeAlternative() bzw. executeFolding() übersetzt.

Test von Invarianten

Zum Ende dieses Abschnitts beschreiben wir die Übersetzung des Laufzeittests von potentiellen Invarianten, der auf S. 28 f. eingeführt wurde. Das nachfolgende GANILA-Programmfragment Example5 zeigt den Quellcode eines elementaren Sortierverfahrens: Sortierung durch Einfügen (Insertion Sort).

```
ganila class Example5 {
  int A[] = new int[]{ 12, 8, 10, 2 };
  public void start(String[] argv) {
    for(int i = 1, j, v; i < A.length; i++) {                    // PP 0
      // Invariante I: A ist von Index 0 bis i − 1 aufsteigend sortiert
      *IV(isSorted(0, i − 1)) *{                                 // PP 1
        v = A[i];                                                // PP 2
        j = i;                                                   // PP 3
        while(j > 0 && A[j − 1] > v) {                           // PP 4
          // Invariante II: v ≤ A[k] für k = j bis i
          A[j] = A[j − 1];                                       // PP 5
          j−−;                                                   // PP 6
        }
        A[j] = v;                                                // PP 7
      *}
    }
  }
  interactive boolean isSorted(int start, int end) {
    // Test, ob das (globale) Feld A von Index 0 bis i − 1 aufsteigend sortiert ist
    // Liefert Booleschen Wert als Ergebnis zurück
  }
}
```

Die Implementierung besteht aus zwei ineinander geschachtelten Schleifen und erhält ein Feld A mit Zahlenwerten als Eingabe. Die erste Schleife wählt alle Elemente aus A nacheinander aus und fügt diese an den richtigen Positionen der bereits ausgewählten (und sortierten) Elemente ein. Der Prozeß des Einfügens wird von der zweiten Schleife über Rechtsverschiebungen der Feldelemente vorgenommen. Bei genauerer Betrachtung kann man zwei (hypothetische) Invarianten erkennen. Invariante I bezieht sich auf die äußere **for**-Schleife. Sie besagt, daß das Eingabefeld von Index 0 bis zu Index $i - 1$ aufsteigend sortiert ist, und wird zur Laufzeit durch das GANILA-Konstrukt ***IV** überprüft. Dazu wird eine Methode isSorted an jedem der PPs 2 bis 7 aufgerufen. Diese Methode kontrolliert die Gültigkeit der Invariante I an diesen Programmpunkten. Invariante II wurde als Kommentar im Programmbeispiel hinzugefügt. Wir werden sie aus Platzgründen nicht weiter betrachten.

Die Übersetzung nach JAVA wird nach den aus den vorherigen Beispielen bekannten Schemata vorgenommen. Die Invariantenannotation wird wieder als eigener PP (hier mit der Nummer 1) aufgefaßt und in den Aufruf einer Methode executeInvariant() transfor-

miert. Der Compiler GAJA erzeugt zwei zusätzliche Methoden: execute_1() für die Repräsentation des von *{ und *} umschlossenen Blocks und check_1() für die Überprüfung der Hypothese. Wir geben zur Vereinfachung nur die Übersetzung der Programmpunkte 0 und 1 an:

```
public class Example5Algorithm extends GAlgorithm {
  Gint A[] = new Gint[]{ new Gint(12), new Gint(8), new Gint(10), new Gint(2)};
  public void start(String[] argv) {
    rts.enterBlock(); rts.put(argv, "argv");
    dispatch(0, null);
    rts.exitBlock();
  }
  public void dispatch_0() {
    rts.enterBlock();
    Gint i = new Gint(1), j = new Gint(), v = new Gint();
    rts.put(i, "i"); rts.put(j, "j"); rts.put(v, "v");
    for( ; i.getValue() < A.length; i.postInc() ) {
      dispatch(1, new Object[]{ i, j, v });
    }
    rts.exitBlock();
  }
  public ControlFlow dispatch_1(Gint a1, Gint a2, Gint a3) {
    return executeInvariant(1, new Object[]{ a1, a2, a3 });
  }
  public ControlFlow execute_1(Gint a1, Gint a2, Gint a3) {
    dispatch(2, new Object[]{ a1, a3 });
    dispatch(3, new Object[]{ a1, a2 });
    dispatch(4, new Object[]{ a2, a3 });
    dispatch(7, new Object[]{ a1, a2 });
    return new ControlFlow();
  }
  public boolean check_1() {
    Gint i = (Gint) rts.get("i");
    return isSorted(new Gint(0), new Gint(i.getValue() - 1));
  }
}
```

Die im Beispielcode auftretende Variable rts referenziert ein Objekt der Klasse RuntimeStack des GANIMAL-Laufzeitsystems. Mit Hilfe dieser zur Laufzeit der Algorithmenanimation gebildeten Symboltabelle ist es möglich, die generierte Testmethode check_1() an den Programmpunkten 2 bis 7 mit dem richtigen Wert der lokal definierten Variablen i im Methodenaufruf isSorted(0, i − 1) auszuführen. Handelt es sich um globale Klassenvariablen, wie etwa das Feld A, dann kann auf diese direkt zugegriffen werden. Hierbei erzeugt GAJA nur dann den Programmcode für die Steuerung dieser Symboltabelle, wenn mindestens ein ***IV**-Konstrukt im Eingabeprogramm vorhanden ist.

Während der Generierung überprüft GAJA, ob alle in der Hypothese enthaltenen Variablen für den aktuellen Block deklariert sind. Wenn nicht, wird eine Fehlermeldung ausgegeben. Gibt der Visualisierer eine mit dem Modifikator **interactive**[4] versehene Methode an, dann kann diese aus der Methode check_1() heraus aufgerufen werden. Zu beachten ist, daß interaktive Methoden in ihrer Übersetzung keine reifizierten Programmpunkte enthalten dürfen. Interaktive Methoden werden von GAJA fast unverändert übersetzt. Lediglich Variablen mit primitiven Typen werden wie bisher in ein entsprechendes Objekt verpackt. Ferner dürfen sie auch keine Methodenaufrufe zu nicht-interaktiven Methoden des Algorithmenmoduls enthalten, können aber auf Klassenvariablen zugreifen. Sie stellen sozusagen atomare Einheiten dar, die auf einmal ausgeführt, aber nicht animiert werden.

Unter Berücksichtigung der PPEs von Programmpunkt 1 des Invariantenkonstrukts ruft die von der Klasse GAlgorithm geerbte Methode executeInvariant() beim Betreten und Verlassen des Invariantenblocks zwei weitere Methoden des Laufzeitsystems auf:

```
public class GAlgorithm {
  ...
  public synchronized ControlFlow executeInvariant(int pp, Object[] args) {
    ...
    if(st[pp].isIVActive()) { active = true; }
    if(active) openIV(pp);
    // Berechne die Variable c und das Feld parameters[] wie zuvor
    Method m = c.getMethod("execute_" + pp, parameters);
    Object o = m.invoke(this, args);
    if(active) closeIV(pp);
    ...
  }
}
```

Die beiden Methoden openIV() und closeIV(*pp*) verwalten einen Keller, der mehrere jeweils ineinander geschachtelte Invariantenkonstrukte aufnimmt. Er findet ebenfalls Verwendung, wenn infolge einer Rekursion mehrere lebende Instanzen einer Invariante existieren. Weiterhin verschicken die zwei Methoden Events an eine sog. Invariantensicht (vgl. Abschnitt 5.4.2 auf S. 95), wie im folgenden erläutert wird:

- openIV(*pp*): Lädt den PP auf den Keller und schickt ein systemgeneriertes Event IE_Start(*pp*, *ivString*) an die Invariantensicht. Diese stellt die ID und den Hypothesentext in einer graphischen Darstellung des Kellers dar.
- closeIV(): Entfernt das oberste Kellerelement, d. h. die Informationen des innersten Invariantenkonstrukts, und schickt ein systemgeneriertes Event IE_Close() an die Invariantensicht.

[4] „Interaktive“ Methoden, da jeder im Invariantenblock enthaltene (reifizierte) Programmpunkt eine solche Methode aufrufen kann.

Solange der Keller nicht leer ist, sendet unsere generische dispatch()-Methode an jedem abzuarbeitenden PP eine Liste mit Wahrheitswerten über ein spezielles Event ***IE_CheckPP(*BoolListe*) an die Invariantensicht. Zum Aufbau der Liste wird für jede auf dem Keller gespeicherte Invariante mit der Nummer *pp* über den Aufruf der generierten Methode check_*pp*() ein Boolescher Wert ermittelt. Auf diese Weise wird festgestellt, an welchen Programmpunkten die Invariante verletzt wird oder erhalten bleibt.

3.2.5 Verwandte Arbeiten und Techniken

Im folgenden beleuchten wir kurz einige Methoden und Arbeiten, welche in die Entwicklung der hier vorgestellten Reifikationstechnik in unterschiedlicher Ausprägung mit eingeflossen sind.

Reflexion

Die *Java Reflection API* [CWH99] bietet Mechanismen, um Klassen zu laden, zu inspizieren und zu instantiieren sowie auf ihre Methoden und Variablen zuzugreifen, selbst wenn die Namen und Signaturen dieser Klassen, Methoden und Variablen zur Übersetzungszeit nicht bekannt sind. JAVA-Reflection unterstützt aber keinen Zugriff auf Programmpunkte zur Laufzeit.

JAVA-Reflection alleine ist somit für unsere Zielsetzung nicht geeignet, wird jedoch zur Implementierung der Reifikation an vielen Stellen des Frameworks verwendet.

JPDA

Die *Java Platform Debugger Architecture* JPDA [Mic02] besteht aus einer API, einem Kommunikationsprotokoll und einer nativen Schnittstelle, die JVM[5]-Implementierungen zur Fehlersuche nutzen können. Durch die Verwendung von JPDA erhält ein Debugger Informationen, wie etwa den aktuellen Kellerrahmen oder den Bytecode von Methoden, setzt Abbruchpunkte auf der Ebene von Bytecode-Instruktionen und registriert bestimmte Ereignisse. Die JPDA-Architektur wird gegenwärtig nicht weitreichend durch JVMs unterstützt, geschweige denn von JIT[6]-Compilern oder anderen Compilern, die plattformabhängigen Code erzeugen.

JPDA wird im GANIMAL-Framework daher nicht genutzt. Es existieren jedoch andere Systeme zur Programmvisualisierung, die JPDA verwenden, z. B. das JAVAVIS-System von Oechsle und Schmitt [OS02].

Reifikation

D. P. Friedman und M. Wand grenzen die beiden Begriffe „Reification“ und „Reflection“ in [FW84] folgendermaßen voneinander ab:

[5] JVM – Java Virtual Machine
[6] JIT – Just In Time

> „We will use the term *reification* to mean the conversion of an interpreter component into an object which the program can manipulate. One can think of this transformation as converting program [...] into data. We will use the term *reflection* to mean the operation of taking a program-manipulable value and installing it as a component in the interpreter. This is thus a transformation from data to program."

Es gibt in der Literatur mehrere Ansätze zur Reifikation, die auf sehr unterschiedlichen Techniken beruhen: Douence und Südholt [DS01] schlagen eine generische Reifikationstechnik vor, die auf Programmtransformationen basiert. Ihre Methode verlangt die Verfügbarkeit des Interpreterquellcodes für die entsprechende Sprache und erlaubt, verschiedene Teile des Interpreters selektiv zu reifizieren; vermutlich auch solche Teile, die einzelne Programmpunkte interpretieren.

Lawall und Muller [LM00] erweitern Programme um Aufrufe einer generischen Funktion checkpoint(), welche den aktuellen Zustand bestimmter Objekte zur späteren Fehlerentdeckung bzw. für post mortem-Untersuchungen abspeichern. Um eine Leistungsverbesserung zu erzielen, verwenden sie eine automatische Programmspezialisierung. Diese generiert spezielle Überwachungspunktroutinen, die nur eine Teilmenge aller gespeicherten Daten durchlaufen. Sie wird mit Hilfe statischer Analysen bestimmt und enthält genau die Daten, die seit dem letzten Überwachungspunkt modifiziert worden sind.

Die Funktion checkpoint() hat gewisse Ähnlichkeit mit der in unserem Ansatz verwendeten, allgemeinen dispatch()-Methode, welche die für jeden Programmpunkt erzeugte Methode zur Laufzeit des Algorithmus über Java-Reflection aufruft.

3.3 Zusammenfassung

Unsere Reifikationstechnik erlaubt es, zur Laufzeit auf einzelne Programmpunkte zuzugreifen. Dazu ist keine spezielle JVM-Implementierung erforderlich. Beliebige Metainformationen können mit diesen Programmpunkten assoziiert und zur Laufzeit manipuliert werden. Das Ausführungsverhalten läßt sich durch die Veränderungen der generischen dispatch()-Methode erweitern oder modifizieren. Wir diskutierten mehrere Erweiterungen, die wir für die visuelle Ausführung von Java-Programmen verwendet haben, vor allem die Mischung von post mortem- und live/online-Visualisierung und die kontrollierte Visualisierung von Schleifen bzw. Rekursionen.

Die in den Abschnitten 3.2.2 und 3.2.3 präsentierte Reifikationsmethode könnte aber auch für andere Anwendungsbereiche genutzt werden. Zum Beispiel könnten wir eine weitere Indirektionsebene einführen und eine Tabelle PP erhalten, welche Zahlen zu Programmpunkten zuordnet. Dann würde die generische Methode dispatch(pp, ...) in ihrem Rumpf die Methode dispatch_pp'(...) mit $pp' = \mathsf{PP}[pp]$ aufrufen. Durch Veränderung der Tabelleneinträge, z. B. mit Hilfe einer graphischen Benutzerschnittstelle, könnte der Benutzer zur Laufzeit ein Programm aus Fragmenten konstruieren. Dies wäre für einen konstruktivistischen Ansatz — der Begriff des Konstruktivismus wird in Kapitel 7 detailliert erklärt — in der Algorithmenlehre nützlich.

Daneben ist die obige Anwendung der reifizierten Programmpunkte zur visuellen Ausführung auch für Softwaretechniker interessant. Da Reflexion der Schlüsselmechanismus für die spätere Komposition von Component Software [Szy98] ist, stellt die Bedeutung reifizierter Programmpunkte in diesem Kontext ein interessantes neues Forschungsfeld dar.

4 Der Compiler GAJA

GaJa generiert aus einem gegebenen Ganila-Programm mehrere Module, die gemeinsam mit den Komponenten der Laufzeitumgebung interaktive Animationen des Eingabeprogramms ermöglichen. Die erzeugten Klassen und Dateien verwenden eine Reihe von weiteren Klassen, Schnittstellen und Methoden der Laufzeitumgebung von Ganimal, so daß es öfter notwendig ist, auf Abschnitte vorangegangener Kapitel und auf die des nachfolgenden Kapitels über die Ganimal-Laufzeitumgebung zu verweisen.

Algorithmenmodul In Abschnitt 3.2 wurde eine Technik vorgestellt, die alle im Ganila-Eingabeprogramm auftretenden Programmpunkte durch die Übersetzung in spezielle Java-Methoden reifiziert. Das Algorithmenmodul wird durch eine neue Klasse mit diesen Methoden als Instanzmethoden implementiert. Sie erbt von der Klasse GAlgorithm des Laufzeitsystems und führt den zu animierenden Algorithmus aus.

Initiale PPEs Für jeden reifizierten Programmpunkt werden seine initialen Einstellungen durch eine zweite Klasse in Java kodiert. Programmpunkte, die einen Moduswechsel bzw. Haltepunkt repräsentieren, bekommen ein entsprechendes PPE-Objekt zugewiesen. Das Gleiche gilt für alle vorkommenden Ganila-Konstrukte und Schleifen mit einer Visualisierungsbedingung. Per Voreinstellung sind sämtliche PPEs beim Start der Animation aktiviert und können über die GUI manipuliert werden.

Abstrakter Syntaxbaum Dieselbe generierte Klasse enthält einen Konstruktor für den Aufbau des abstrakten Syntaxbaums (AST) des Eingabeprogramms. Der Syntaxbaum wird sowohl von der CodeView (vgl. Abschnitt 5.4) als auch von der GUI benutzt. Beide Komponenten zeigen den Programmcode an und benötigen den Syntaxbaum zur Strukturierung der Programmcodedarstellung. Zur vereinfachenden Darstellung der Zusammenhänge wird im folgenden davon ausgegangen, daß die initialen PPEs und der AST zwei voneinander getrennte Module darstellen.

Templatesicht GaJa erzeugt eine Klassendatei mit Templatemethoden, von der alle benutzerdefinierten Sichten auf das zugrundeliegende Eingabeprogramm erben müssen. Sie definiert zur Entlastung des Visualisierers für jedes im Ganila-Code enthaltene Interesting Event unterschiedlichen Namens vier verschiedene Eventhandler, wie in Abschnitt 3.2.4 bereits diskutiert wurde. Diese realisieren eine erweiterte post mortem-Visualisierung auf der Grundlage unterschiedlicher Animationsmodi und Aktivierungszustände der IEs. Die generierte Klasse implementiert das Entwurfsmuster *Template*

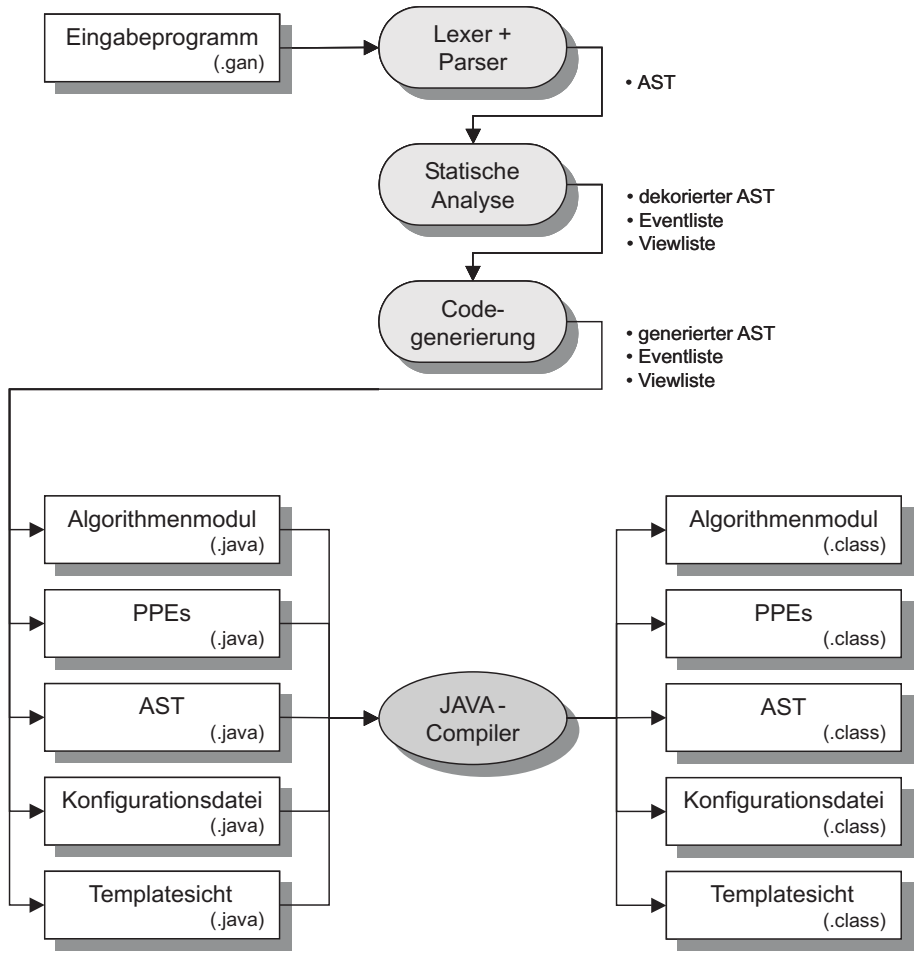


Abbildung 4.1: Überblick über die Architektur des Compilers GAJA.

Method [GHJV95] und ermöglicht so zusätzlich, invariante Operationen benutzerdefinierter Sichten in einer separaten Klasse zusammenzufassen.

Konfigurationsdatei Schließlich wird noch eine Konfigurationsdatei für das Laden der erzeugten Klassendateien über die graphische Benutzerschnittstelle erstellt. Sie enthält z. B. Informationen über den in GANILA spezifizierten Paketnamen, den Namen des Algorithmus für den Animationstitel oder den Übersetzungszeitpunkt.

4.1 Architektur

Der zugrundeliegende Aufbau des Compilers GAJA folgt im Frontend klassischen Entwurfsprinzipien des Übersetzerbaus, wie in Abbildung 4.1 dargestellt ist. Ein in der Animationsbeschreibungssprache GANILA spezifiziertes Eingabeprogramm wird zunächst durch einen Lexer und einen Parser syntaktisch analysiert. Während dieser Überset-

zungsphase wird ein abstrakter Syntaxbaum (AST) erzeugt, welcher als Eingabe der nachfolgenden Phase der statischen Analyse dient. Diese berechnet bestimmte semantische Eigenschaften des Eingabeprogramms, wie z. B. Ausdruckstypen oder Identifizierung von Bezeichnern. Sie erstellt u. a. Listen der im Programm auftretenden Events sowie der eingebundenen Sichten und liefert einen mit diesen semantischen Informationen dekorierten AST zurück. Der Codeerzeuger bildet aus diesem AST nicht direkt neuen Quellcode, sondern einen zweiten AST, der einem reinen JAVA-Programm entspricht. Schließlich erzeugt eine letzte Phase aus dem neuen AST und den Listen die oben diskutierten fünf Module in Gestalt von JAVA-Dateien. Diese werden anschließend mit Hilfe des Compilers JAVAC in Bytecode übersetzt. JAVAC ist der Standardcompiler der von der Firma Sun bereitgestellten Entwicklungs- und Laufzeitumgebung J2SE™ (JAVA 2 Platform Standard Edition).

In den nachfolgenden Abschnitten werden die Generierung des GANILA-Parsers, die statische Analyse sowie die Codeerzeugung tiefgreifender diskutiert. Aus softwaretechnischem Blickwinkel betrachtet, sind sämtliche Analyse- und Berechnungsschritte des Compilers GAJA gemäß dem sogenannten *Visitor Pattern* (Besuchermuster) strukturiert. Eine Ausnahme ist der Parser, der mit Hilfe eines frei verfügbaren Werkzeugs (s. u.) erzeugt wurde. Wir erläutern kurz die Grundidee dieses Entwurfsmusters und motivieren dessen Realisierung in unserem Übersetzer.

Visitor Pattern

Die Intention für den Gebrauch des Besuchermusters ist die Abkapselung von Operationen auf einer Datenstruktur von deren Elementen. Auf diesem Weg kann man eine Operation verändern oder neue Operationen hinzufügen, ohne die Klassen der Elemente selbst modifizieren zu müssen, auf denen die Operation angewendet wird. Um dieses Ziel zu erreichen, definiert das Muster eine Zweiklassenhierarchie: eine für die Elemente, auf denen operiert wird, und eine für die Besucher, welche Operationen auf den Elementen definieren [PJ98, GHJV95]. Aufgrund dieser Klassenhierarchien reicht es aus, einen neuen Besucher der eigenen Hierarchie hinzuzufügen, um eine neue Operation zu implementieren (siehe Abb. 4.2).

Jedes Element der Datenstruktur *akzeptiert* einen Besucher über die Definition einer Methode `accept(Visitor v)` durch die Versendung einer Nachricht zu einem Besucher `v`. Diese Nachricht beinhaltet die Klasse des Elements und das Element selbst. Falls die zugrundeliegende Programmiersprache Methodenüberladung kennt, muß der Klassentyp des Elements nicht explizit angegeben werden. Ein Besucher `v` wird dann seine spezielle Operation für genau dieses Element ausführen. Dieser Prozeß ist als *Double Dispatching* bekannt. Die Ausführung der `accept()`-Methode hängt nicht nur vom Elementtyp ab, sondern auch vom Typ des Besuchers. Die Bindung der Operation an das entsprechende Element geschieht hier zur Laufzeit des Programms. Abstrakte Klassen der Besucher und der Elemente stellen die Spitze der jeweiligen Hierarchie dar. Die abstrakte Elementklasse enthält die o. g. `accept()`-Methode und die abstrakte Besucherklasse eine Methode `visit_`*ElType*`()` für jeden Elementtyp *ElType*. Bei Methodenüberladung wird eine Methode `visit()` einfach mit dem jeweiligen Elementtyp überladen, so daß deren Aufruf implizit

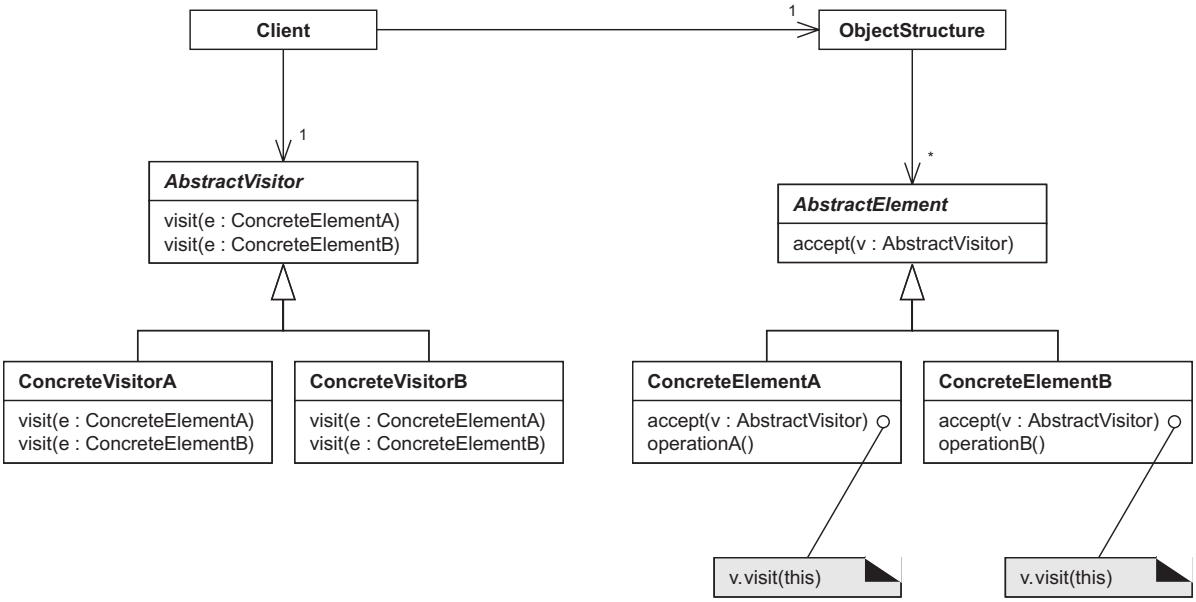


Abbildung 4.2: Struktur des Besuchermusters (UML-Notation, vgl. [Oes97]). Die Darstellung der Klassenoperationen setzt eine Programmiersprache mit Methodenüberladung voraus.

erfolgt. Konkrete Elementklassen überschreiben die accept()-Methode, um ihr eigenes Objekt samt Typ an einen Besucher zu übermitteln, konkrete Besucher überschreiben nur die visit()-Methoden eines Elementtyps, für den sie eine Operation definieren.

Algorithmen auf Syntaxbäumen sind ein klassisches Beispiel für die Verwendung des Besuchermusters. Elemente sind die Knoten des Syntaxbaums, und Besucher entsprechen den verschiedenen Operationen, die auf dem Baum ausgeführt werden, z. B. Typechecking, Codeoptimierung, Codeerzeugung und -ausgabe. So ist es einfach, etwa alle Knoten von Methodendeklarationen im AST zu zählen: Man implementiert einen Besucher, welcher von einem abstrakten Visitor erbt, und schreibt eine einzige visit()-Methode für den Knotentyp MethodDecl. Diese enthält einen global definierten Zähler, der mit jedem Aufruf der Methode inkrementiert wird. Aufgrund dieser Vorteile wurden Besuchermuster für die Implementierung des Compilers GAJA verwendet.

Besuchermuster sind allerdings nachteilig, falls sich die zugrundeliegende Datenstruktur häufig ändert und neue Elementtypen hinzukommen. Dann müssen im schlimmsten Fall alle Besucher aufwendig an die veränderte Situation angepaßt und neue visit()-Methoden implementiert werden. In der Entwicklung eines Compilers kommen Modifikationen in der Grammatik der zu übersetzenden Sprache allerdings seltener vor und wenn, dann eher zu Beginn der Implementierung.

4.2 Generierung des Parsers

Die Parsergenerierung vollzieht sich in zwei Schritten: Aus einer gegebenen EBNF-Grammatik für GANILA erzeugt das frei verfügbare Werkzeug *Java Tree Builder* (JTB 1.2.2, [PTW02]) die folgenden Komponenten:

- eine Menge von Klassen für die verschiedenen Knotentypen des AST, die auf den Produktionen der Eingabegrammatik basieren und das Visitor Pattern unterstützen,
- mehrere Schnittstellen und vordefinierte Besucher, deren visit()-Methoden die Kinder des jeweils aktuellen Knotens per DFS besuchen, und
- eine neue EBNF-Grammatik mit zusätzlichem JAVA-Code, um den Syntaxbaum während des Parsings aufzubauen.

Neue Besucher können nun von den vordefinierten Besuchern erben und deren visit()-Methoden (teilweise) überschreiben. Somit können in späteren Übersetzungsphasen verschiedene Operationen und Algorithmen auf dem generierten Syntaxbaum durchgeführt werden.

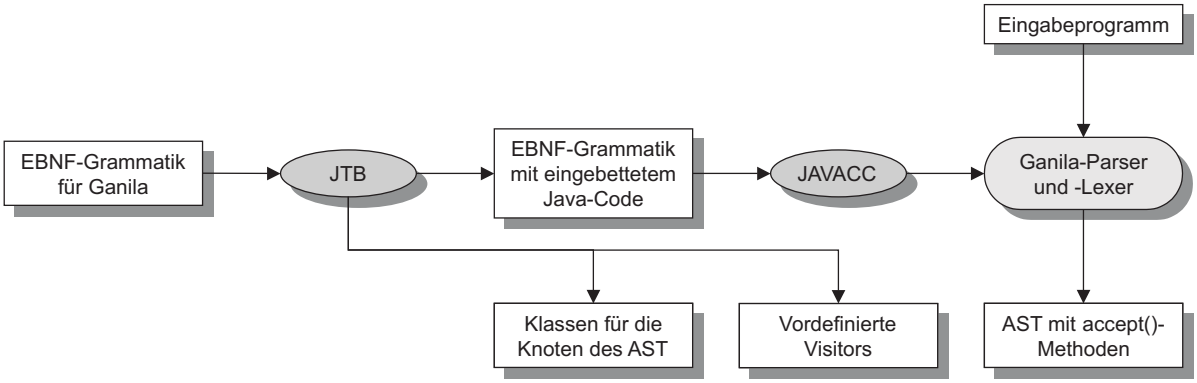


Abbildung 4.3: Überblick über die Generierung des GANILA-Parsers.

Die eigentliche Generierung des Parsers übernimmt dabei WebGain's *Java Compiler Compiler*^{TM} (JAVACC 2.1, [Web02]). Dieses Werkzeug erhält die um JAVA-Code erweiterte Grammatik als Eingabe und generiert einen LL(1)-Parser und einen Lexer für unsere Programmiersprache GANILA. Jede rechte Seite (RHS) einer Grammatikproduktion wird in eine Methode des erzeugten Parsers, und jede linke Seite (LHS) in einen entsprechenden Methodenaufruf transformiert. Im Ablauf der syntaktischen Analyse bauen diese Methoden zusammen mit dem von JTB hinzugefügten Code einen AST auf, dessen Knotenklassen accept()-Methoden zur Realisierung des Besuchermusters bereitstellen. Abbildung 4.3 verdeutlicht diese Zusammenhänge graphisch.

4.3 Statische Analyse

Nachdem der AST vollständig erzeugt wurde, führt unser Übersetzer mehrere Analysen auf diesem Syntaxbaum aus. Die Analyseergebnisse werden an den entsprechenden Knoten des AST abgespeichert und in der Synthesephase zur Generierung des Ausgabeprogramms verwendet. GANILA ist eine Erweiterung der Programmiersprache JAVA und ermöglicht damit auch Variablendeklarationen an beliebiger Stelle sowie Vorwärtsreferenzen auf Methodendefinitionen. Das heißt Methoden können aufgerufen werden, bevor sie definiert sind. Diese und weitere Eigenschaften erlauben es nicht, die Analysen in einem Lauf über den AST zu bewältigen. Deshalb unterteilen wir die Analysen in mehrere Stufen: Zunächst werden globale Eigenschaften der Klasse berechnet, wie z. B. die Erzeugung der o. g. Listen für die Sichten und Interesting Events sowie Typumgebungen für Methoden und Felder der Klasse. Danach folgt die Berechnung lokaler Eigenschaften innerhalb der Definitionen von Methoden und Feldern, etwa die Verarbeitung von lokalen Variablendeklarationen, die Typberechnung von Ausdrücken etc.

4.3.1 Berechnung von globalen Klasseneigenschaften

Die Implementierung der ersten Analysestufe wurde über ein Besuchermuster realisiert. Sie weist zunächst den Knoten des AST, die einen Programmpunkt des zugrundliegenden Eingabeprogramms repräsentieren, jeweils eindeutige Kennummern $pp \in PPM$ mit $PPM = \mathbb{N}_0$ zu. Weiterhin werden Paket- und Importinformationen gesammelt. Für die spätere Generierung der Templateklasse und die Registrierung der deklarierten Sichten werden Listen der im Programm enthaltenen Interesting Events bzw. der in der Sichtendeklaration angebenen Informationen erstellt.

Schließlich erzeugt GAJA eine Typumgebung für in der Klasse enthaltene Felder (Variablendeklaration und deren Initialisierung) sowie für Methodendefinitionen. Bei dieser Gelegenheit werden auch interaktive Methoden entsprechend berücksichtigt. Im folgenden diskutieren wir die Struktur und Verwaltung der Typumgebung etwas genauer.

Typumgebung

In Abschnitt 3.2 zur Reifikation von GANILA-Programmen wurde beschrieben, daß der Übersetzer für jeden Programmpunkt eine neue Methode erzeugt und diese an der Stelle seines ursprünglichen Vorkommens aufruft. Um lokal definierte Variablen als Argumente an diese generierten Methoden zu übergeben, wurden für alle primitiven Typen von GANILA eigene Wrapperklassen definiert. Abbildung 4.4 zeigt die Typhierarchie von GANILA. Sie entspricht der in [GJSB00] definierten Typhierarchie von JAVA, mit Ausnahme der Typen, die mit einer gestrichelten Kante verbunden sind. Alle in der Abbildung fett gedruckten Typen sind in unserem Framework durch eine Wrapperklasse ersetzt worden.

Betrachtet man ein angewandtes Vorkommen einer im Eingabeprogramm lokal definierten Integervariablen x, z. B. innerhalb einer Zuweisung mit Programmpunkt *pp*, dann taucht dieses im erzeugten Programmcode in einem anderen Kontext unter anderem Namen, z. B. a3, wieder auf: im Rumpf einer neu generierten Methode dispatch_*pp*(. . . , Gint

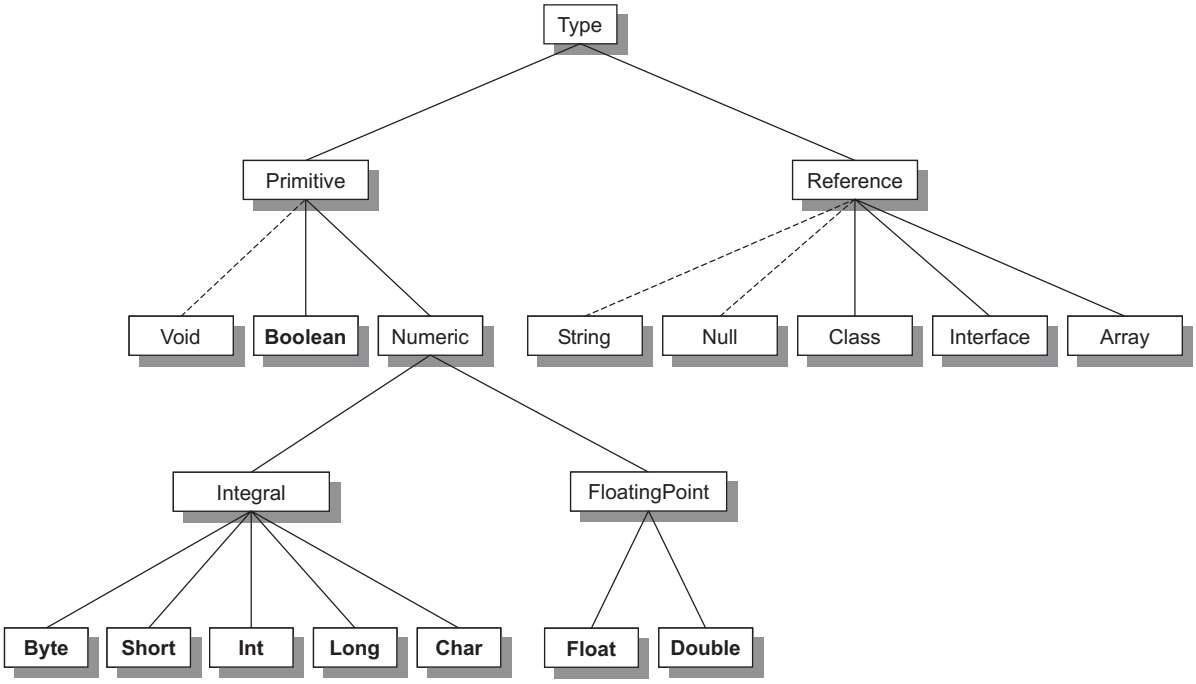


Abbildung 4.4: Typhierarchie von Ganila.

a3, ...). Daher muß unsere Typumgebung für Variablen nicht nur den Typ, die Position (`global`, `local`) sowie den Gültigkeits- und Sichtbarkeitsbereich verwalten, sondern zusätzlich neue *Übergabevariablen*, die von der Tiefe der sich gegenseitig aufrufenden Dispatchmethoden[1] abhängen. Von Seiten der Implementierung wird dies durch eine erweiterte Symboltabelle erreicht, die für jeden Eintrag neben den üblichen Informationen (vgl. hierzu das Lehrbuch [WM96]) noch einen zusätzlichen Keller zur Namensverwaltung der Übergabevariablen aufweist. Wir definieren unsere Typumgebung ρ wie folgt:

$$\rho : \mathit{Name} \rightarrow (\mathit{Type} \times \{\texttt{global}, \texttt{local}\}) \times (\mathit{Name} \cup \{\varepsilon\}) \tag{4.1}$$

Treffen wir in unserer Analyse beispielsweise auf die Deklaration einer Integervariablen x, so wird eine neue Variablenbindung erstellt und $\rho(\texttt{x}) = ((\texttt{int}, \texttt{global}), \varepsilon)$ gesetzt. Global definierte Variablen- und Methodennamen werden in unserer Übersetzung beibehalten[2].

[1] Der Begriff „Dispatchmethode“ wird im folgenden zur besseren Lesbarkeit synomym für „dispatch()-Methode“ verwendet.

[2] In der aktuellen Version unterstützt GaJa noch keine Methodenüberladung. Methodensignaturen bleiben zwar in unserer Übersetzung bis auf primitive Typen unverändert, aber man kann nicht die eingebaute Überladungsauflösung des JAVAC verwenden, da wir in der Übersetzung von Ausdrücken den Rückgabetyp von Methodenaufrufen berechnen müssen. Dazu ist die Typumgebung um Methodensignaturen zu erweitern und ein Algorithmus zu implementieren, der die passendste Methode bei einem Aufruf auswählt. Ferner müssen Methoden und Felder anderer Klassen explizit über Casts konvertiert werden.

4.3.2 Berechnung lokaler Eigenschaften von Methoden und Feldern

Nachdem die Typen klassenglobaler Konstrukte in den entsprechenden Knoten des AST abgespeichert und deren Bindungen in die Typumgebung ρ eingetragen worden sind, analysiert die zweite Stufe lokale Eigenschaften von Methoden und Feldern. Welche Eigenschaften dazu gehören, wird in diesem Abschnitt dargelegt. Unsere Implementierung zum Analyseverfahren der zweiten Stufe ist ebenfalls nach einem Besuchermuster strukturiert. Sie ruft jedoch zur Lösung von Teilaufgaben weitere eigenständige Besucher auf. So existiert ein eigener Besucher für die Durchführung der Typinferenz auf Ausdrücken, der Kontrollflußanalyse, der Suche aller in einer Anweisung enthaltenen Variablennamen, die außerhalb der Anweisung definiert wurden usw.

Wir beschreiben informal die wichtigsten Aufgaben dieser Analysestufe sowie deren Lösungen der Reihe nach und definieren mehrere Umgebungen, die das Analyseergebnis beinhalten und an den entsprechenden Knoten des AST abgespeichert werden. Sie finden Anwendung in der in Abschnitt 4.4 geführten Diskussion zur Codeerzeugung.

Lokale Definitionen und formale Parameter

Die Typumgebung wird bei Deklarationen von lokalen Variablen und formalen Parametern von Methoden — in ähnlicher Weise wie bereits oben beschrieben — um neue Bindungen erweitert. Diese neuen Bindungen, etwa für eine lokale Boolesche Variable `b`, enthalten anfangs keine neuen Variablenbenennungen: $\texttt{b} \mapsto ((\texttt{boolean}, \texttt{local}), \varepsilon)$. Neue Variablenbezeichner werden erst während der Analyse für solche Knoten abgespeichert, die einem Programmpunkt entsprechen und später eigene Dispatchmethoden erhalten.

Ausdrücke

Ein Typinferenzalgorithmus bestimmt den Typ eines Ausdrucks und die Typen aller seiner Teilausdrücke. Da wir in der Generierungsphase alle Variablen primitiven Typs in Objekte der entsprechenden Wrapperklasse verpacken, müssen wir u. a. folgende Gegebenheiten berücksichtigen: Eingebaute Operatoren (**+**, −, **&&**, ...) verlangen R-Werte als Operanden, und Aufrufe von externen Methoden dürfen keine gewrappten Variablen (L-Werte) als Argumente erhalten. Formale Parametertypen von Methoden der reifizierten Klasse werden jedoch durch Wrappertypen ersetzt, so daß L-Werte übergeben werden können. Zur Übergabe müssen diese gewrappten Variablen hier aber kopiert werden, damit keine unerwünschten Seiteneffekte entstehen. Aus diesem Grund stellt unsere Analyse Ausdrücke und Teilausdrücke in einen Kontext, der besagt, ob diese verpackt, geklont oder entpackt werden sollen. Der Kontext ist leer, wenn der betreffende Ausdruck nicht primitiven Typs ist. Eine Umgebung ψ für Ausdrücke kann demnach wie folgt definiert werden:

$$\psi : \mathit{Expr} \rightarrow \mathit{Type} \times \{\texttt{lVal}, \texttt{lCloneVal}, \texttt{rVal}, \varepsilon\} \tag{4.2}$$

Die meisten Operationen auf Operanden primitiven Typs können mit Hilfe der Umgebungen ρ und ψ direkt übersetzt werden. Hierbei bleiben die Operatoren erhalten, d. h.

für diesen Operator wird dasselbe Schlüsselwort generiert. Eine Sonderstellung nehmen allerdings Zuweisungsoperatoren (z. B. =, -= oder <<=) ein, die Seiteneffekte produzieren. Die linke Seite einer Zuweisung kann hier nicht aus einem L-Wert bestehen, da wir sonst zur Laufzeit beispielsweise einem Gint-Objekt unmittelbar den R-Wert eines anderen Gint-Objekts zuweisen würden. Da die Programmiersprache JAVA eine streng typisierte Sprache ist, würde der Standardcompiler JAVAC dies jedoch schon zur Übersetzungszeit als Fehler erkennen. Daher ersetzen wir diese Operatoren durch Methodenaufrufe auf den Objekten der linken Seite. Wir werden im nächsten Abschnitt wieder auf diese Problematik zurückkommen.

Umgebung für die Übergabe der lokalen Variablen an die Dispatchmethoden

Entspricht ein Knoten des AST einem zu reifizierenden Programmpunkt, dann erzeugt GAJA, wie bereits erläutert, eine Dispatchmethode für diesen PP und einen entsprechenden Aufruf im Rumpf

- einer bereits erzeugten Dispatchmethode, wenn der aktuelle PP im Block eines anderen PP (z. B. einer Schleife) enthalten ist, oder
- einer Methode der reifizierten Klasse, wenn der aktuelle PP nicht im Rumpf höherer Kontrollkonstrukte enthalten ist.

Für eine korrekte Übergabe lokal definierter Variablen sind an jedem Programmpunkt *pp* eine Reihe weiterer Analysen notwendig, deren Resultate im entsprechenden Syntaxbaumknoten $node(pp)$ abgelegt werden:

Variablenliste In einem ersten Schritt sammelt der Übersetzer alle im Block eines Programmpunkts *pp* enthaltenen lokalen Variablen in einer Liste $Vars(pp)$. Zu einem späteren Zeitpunkt erzeugt er aus dieser Liste die Argumente des Dispatchmethodenaufrufs bzw. die formalen Parameter der Dispatchmethode. Dazu durchläuft er den Unterbaum von $node(pp)$ und speichert alle angewandten Vorkommen von aktuell in ρ enthaltenen lokalen Variablen.

$$Vars : PPM \rightarrow \mathcal{P}(Name) \tag{4.3}$$

Nehmen wir als Beispiel eine bedingte Anweisung **if (***expr***)** $block_1$ **else** $block_2$**;** als aktuellen PP. Dann speichern wir alle lokalen Variablen, die in *expr*, $block_1$ und $block_2$ ein angewandtes Vorkommen haben, aber nicht diejenigen, die in einem der beiden Blöcke lokal definiert werden.

Parameterübergabe Mit Hilfe der Variablenliste $Vars(pp)$ speichern wir für einen Programmpunkt *pp* die Argumente des Dispatchmethodenaufrufs sowie die entsprechenden formalen Parameterdefinitionen in $node(pp)$: Die Listenelemente werden nacheinander durchlaufen. Simultan werden zwei Vektoren gebildet. Der erste Vektor enthält die Argumente. Ist das Listenelement eine Variable mit dem Bezeichner x, dann wird in der

Typumgebung die aktuelle Bindung $\rho(\mathsf{x})$ nachgeschlagen. Enthält diese eine neue Benennung für eine Übergabevariable, dann wird der entsprechende Name in den Vektor eingetragen, sonst der Bezeichner x selbst.

Bisher sind wir nicht darauf eingegangen, wie und zu welchem Zeitpunkt neue Übergabevariablen für eine lokale Variable in die Typumgebung ρ eingetragen werden. Um den zweiten Vektor für die formalen Parameterdefinitionen zusammenzusetzen, wird ein neuer Name für jeden in der Variablenliste enthaltenen Bezeichner $x \in \mathit{Vars}(pp)$ erzeugt, und dann die aktuelle Bindung des Bezeichners x über ρ ermittelt. Die Bindung beinhaltet, wie oben bereits beschrieben, neben Typ und Position noch einen zusätzlichen Namenskeller für die Übergabevariablen. Auf diesen Keller wird nun der neue Variablenname an oberster Stelle abgelegt. Anschließend werden der Typ und der neue Name in den zweiten Vektor eingetragen. Der hier dargelegte Ansatz impliziert, daß die Analyse des Unterbaums von $\mathit{node}(pp)$ nach diesem Vorgang stattfindet, da die in ρ enthaltenen Bindungen sonst noch nicht angepaßt wären. Wir definieren auf Grundlage dieses Ansatzes eine neue Funktion zur Bestimmung der notwendigen Informationen für die Parameterübergabe:

$$\delta(n) : \mathit{PPM} \rightarrow \mathit{Arg}^n \times \mathit{Decl}^n \tag{4.4}$$

Hierbei sei mit $n \in \mathbb{N}_0$ die berechnete Parameteranzahl für die aktuelle Dispatchmethode bezeichnet, mit PPM die Menge der möglichen Kennummern für einen Programmpunkt, mit $\mathit{Arg}^n = \{(x_1, \ldots, x_n) | x_i \in \mathit{Name} \text{ und } i = 1, \ldots, n\}$ der Wertebereich des Argumentvektors und mit $\mathit{Decl}^n = \{((t_1, y_1), \ldots, (t_n, y_n)) | t_i \in \mathit{Type}, y_i \in \mathit{Name} \text{ und } i = 1, \ldots, n\}$ der Wertebereich des formalen Parametervektors.

Zur Verdeutlichung betrachten wir die Zuweisung tmp2 = y; an PP 1 aus dem Programmbeispiel zur Übersetzung des Paralleloperators auf Seite 42. Das Beispiel zeigt sowohl den ursprünglichen GANILA-Code als auch den generierten JAVA-Code. Für diese Zuweisung wurde aus $\mathit{Vars}(1) = \{\mathsf{tmp2}, \mathsf{y}\}$ für die hier vorgestellte Umgebung $\delta(2)(1) = (\mathsf{tmp2}, \mathsf{y})$ $((\mathtt{int}, \mathsf{a1}), (\mathtt{int}, \mathsf{a2}))$ berechnet. Im generierten Code lautet der Dispatchmethodenaufruf also dispatch(1, **new** Object[]{ tmp2, y }) und die Definition der entsprechenden Dispatchmethode **public void** dispatch_1(Gint a1, Gint a2), wenn der primitive Typ **int** durch den Typ seiner Wrapperklasse Gint ersetzt wird.

Umgebung für den Kontrollfluß

In Abschnitt 3.2.3 haben wir die Übersetzung von Kontrollstrukturen diskutiert und Kontrollflußobjekte des Typs ControlFlow für die Behandlung der Verzweigungsanweisungen **break**, **continue** und **return** eingeführt[3]. Im erzeugten Code liefert die Dispatchmethode einer solchen Anweisung ein Kontrollflußobjekt zurück, welches die Anweisung repräsentiert. Es wird von den umschließenden Dispatchmethoden solange weitergereicht, bis es den korrekten Zielblock, beispielsweise die innerste Schleife im Fall

[3] GAJA stellt bisher keine Funktionalitäten für die Unterstützung von Markierungen (Labels) der Konstrukte **continue** und **break** zur Verfügung. Dazu sind die Kontrollflußobjekte durch den Übersetzer zusätzlich mit den PPs der markierten Anweisungen zu versehen und die Codegenerierung ist entsprechend anzupassen.

der **break**-Anweisung, erreicht und in der entsprechenden Dispatchmethode verarbeitet wird.

Die Verarbeitung des Kontrollflußobjekts sowie dessen Weitergabe von Dispatchmethode zu Dispatchmethode hängen vom Kontext ab, in dem sich ein Programmpunkt befindet. Skizzieren wir ein Beispielszenario, in dem sich eine **return**-Anweisung unmittelbar im Rumpf einer Methode m() befindet. Dann muß der Übersetzer dafür sorgen, daß der in der **return**-Anweisung berechnete Wert nach dem Aufruf ihrer Dispatchmethode von m() zurückgeliefert wird. Dazu wird der Wert aus dem Kontrollflußobjekt entnommen und in den richtigen Typ konvertiert. Wäre die **return**-Anweisung im Rumpf einer Schleife enthalten, dann müßte das Kontrollflußobjekt zunächst von der Dispatchmethode der Schleife weitergeleitet werden bis der Zielblock (hier der Methodenrumpf) erreicht wäre.

Diese Vorgänge werden auch von den durch die Kontrollflußobjekte vertretenen Verzweigungsanweisungen **break**, **continue** und **return** beeinflußt. Der Übersetzer berechnet also für jede Anweisung bzw. für jeden Programmpunkt *pp* den Kontext, in dem sich der Programmpunkt befindet, eine Menge für die im Unterbaum von $node(pp)$ enthaltenen Verzweigungsanweisungen und evtl. den Rückgabetyp in *pp* enthaltener **return**-Anweisungen. Wir definieren auf dieser Grundlage für jede Anweisung eine neue Umgebung γ für Kontrollflußinformationen:

$$\gamma : PPM \rightarrow CflContext \times \mathcal{P}(CflOp) \times RetEnv \tag{4.5}$$

Hierbei sei $CflContext = \{\texttt{method}, \texttt{loop}, \texttt{switch}\}$ die Menge der möglichen Kontextinformationen eines Programmpunkts, $CflOp = \{\texttt{break}, \texttt{continue}, \texttt{return}\}$ die Menge der o. g. Verzweigungsanweisungen und $RetEnv = Type \times \{\texttt{s}, \texttt{m}\}$ eine Menge, aus welcher der Rückgabetyp sowie eine Markierung entnommen wird. Die Markierung `s` repräsentiert *sichere* und die Markierung `m` *mögliche* Ausführungen von **return**-Anweisungen innerhalb einer komplexen Anweisung zur Laufzeit des Algorithmus. Wenn im Eingabeprogramm eine Methode mit einem Rückgabewert deklariert ist, dann muß in ihrem Rumpf zur Laufzeit eine **return**-Anweisung einen passenden Wert zurückliefern [GJSB00]. Mit Hilfe der Markierung erzeugt der Übersetzer Programmcode, der dies sicherstellt. Dies wird im nachfolgenden Abschnitt 4.4 erläutert.

Abbildung 4.5 auf Seite 62 zeigt ein AST-Fragment des Programmcodebeispiels Example1 aus Abschnitt 3.2.2. Das Fragment umfaßt den Rumpf einer Methode start. An einigen Knoten sind die Ergebnisse einer statischen Analyse, d. h. die konkreten Belegungen der verschiedenen Umgebungen, abgespeichert.

4.4 Generierung von JAVA-Code

Die in der statischen Analyse gewonnenen Informationen nutzt GaJa zur Generierung der am Kapitelanfang vorgestellten Module. Dazu wird die Implementierung eines Besuchers mit dem dekorierten AST als Eingabe gestartet, die aus diesem einen neuen AST erzeugt. Danach bildet ein sog. *PrettyPrinter* aus dem neuen AST eine Java-Klasse,

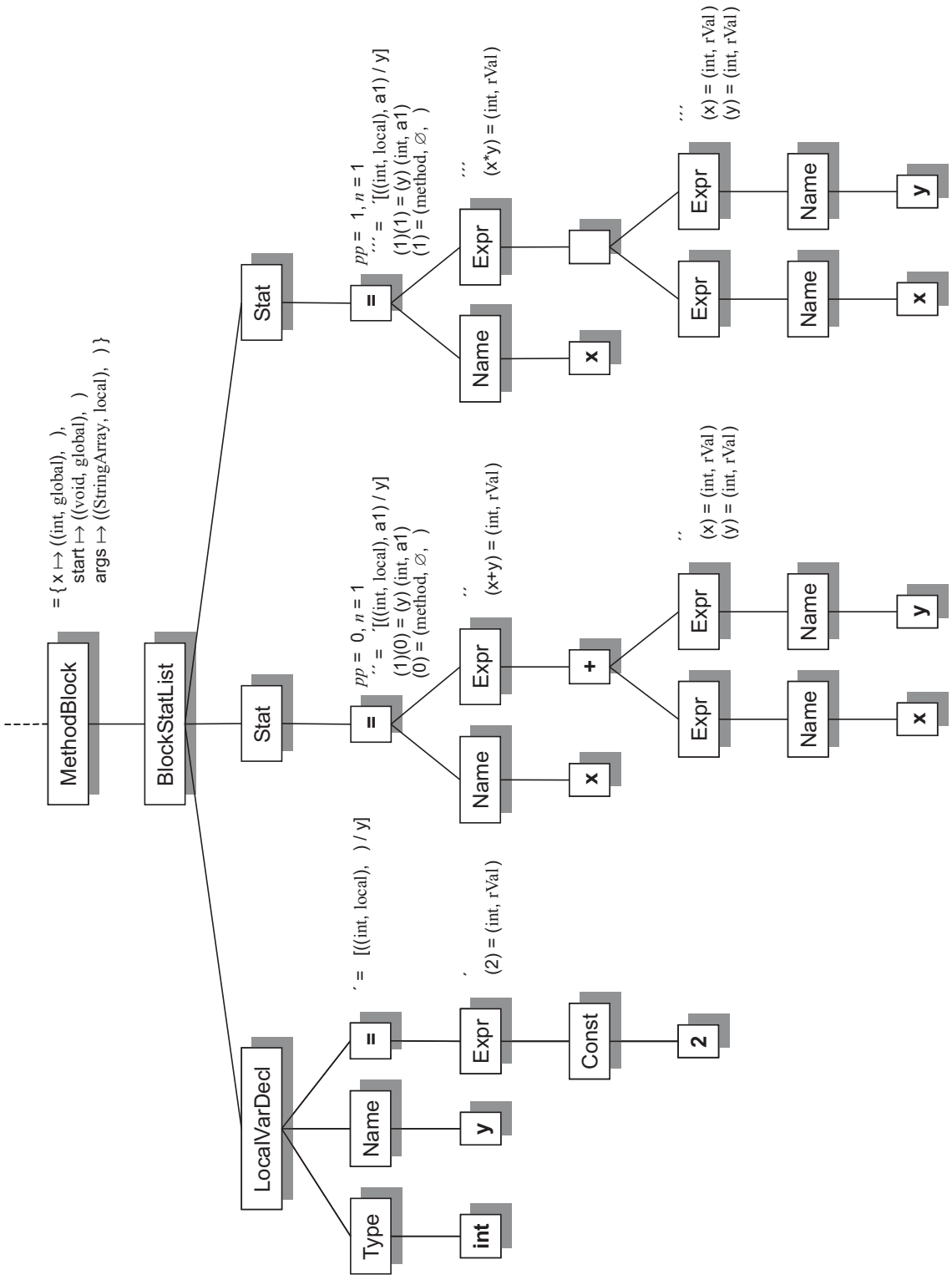


Abbildung 4.5: Auszug aus dem dekorierten AST des Codebeispiels Example1 von S. 33.

die wir in dieser Arbeit mehrfach als Algorithmenmodul bezeichnet haben. GAJA generiert aus den gesammelten Informationen, etwa der Eventliste, zusätzlich noch die vier anderen Module. Um den Umfang dieser Arbeit nicht zu groß werden zu lassen, verzichten wir auf eine Darstellung dieser Prozesse. Stattdessen konzentrieren wir uns auf die Erzeugung des Algorithmenmoduls und beleuchten exemplarisch die Übersetzung einer kleinen Auswahl von Programmkonstrukten: Ausdrücke, Zuweisungen, bedingte Anweisungen sowie **return**-Anweisungen. Wir nehmen im folgenden an, daß die statische Analyse keine Fehler, z. B. Typfehler oder Doppeldeklarationen, aufgedeckt hat, und alle berechneten Informationen an den entsprechenden Knoten des AST vorhanden sind.

Bemerkungen zur Notation

Zur Beschreibung des Übersetzungsvorgangs verwenden wir sog. Codeerzeugungsschemata, wie sie im Lehrbuch von Wilhelm und Maurer [WM96] beschrieben sind. Innerhalb eines solchen Schemas wird dargestellt, wie eine Übersetzungs- oder auch Codeerzeugungsfunktion für ein bestimmtes Sprachkonstrukt einen individuellen Ausgabecode generiert. Diese Funktionen rufen sich i. allg. rekursiv mit einem Teilkonstrukt als neuem Argument wieder auf oder führen weitere Übersetzungs- oder auch Hilfsfunktionen aus. Hilfsfunktionen berechnen hierbei Informationen aus Teilbäumen. Zusätzliche Argumente der Übersetzungsfunktionen propagieren bereits berechnete Informationen zu den Teilkonstrukten.

Wir benötigen für unsere kleine Auswahl von Programmkonstrukten nur zwei Übersetzungsfunktionen: gaja_E für die Übersetzung von Ausdrücken und gaja_M für die Übersetzung von Anweisungen in Methodenrümpfen. Aufrufe und Definitionen von Hilfs- bzw. Übersetzungsfunktionen sind als Boxen

Funktion	*Argumente* bzw. *Parameter*

dargestellt. Diese enthalten den Namen der Funktion und eine Reihe von Argumenten bzw. die formalen Parameter. Generierter Programmcode wird innerhalb eines grauen Rahmens plaziert. Dort können aber auch weitere Aufrufe von Übersetzungs- und Hilfsfunktionen enthalten sein:

Generierter JAVA-Code

In einem Punkt unterscheiden sich unsere Hilfs- und Übersetzungsfunktionen jedoch von solchen, wie sie z. B. in dem o. g. Lehrbuch verwendet werden: Um die Phasentrennung zwischen Analyse und Generierung stärker zu verdeutlichen, operieren die hier beschriebenen Funktionen auf erweiterten ASTs, welche die Ergebnisse der statischen Analyse enthalten, wie u. a. in Abbildung 4.5 gezeigt wurde. Beide Phasen kommunizieren also über den dekorierten AST. Diese Art von Darstellung spiegelt auch unsere Implementierung besser wider. Das heißt im Detail, daß die Argumente bzw. die formalen Parameter von Übersetzungsfunktionen den in den AST-Knoten abgelegten statischen Informationen (Umgebungen) entsprechen, und zur Generierungszeit keinerlei weitere Analysen

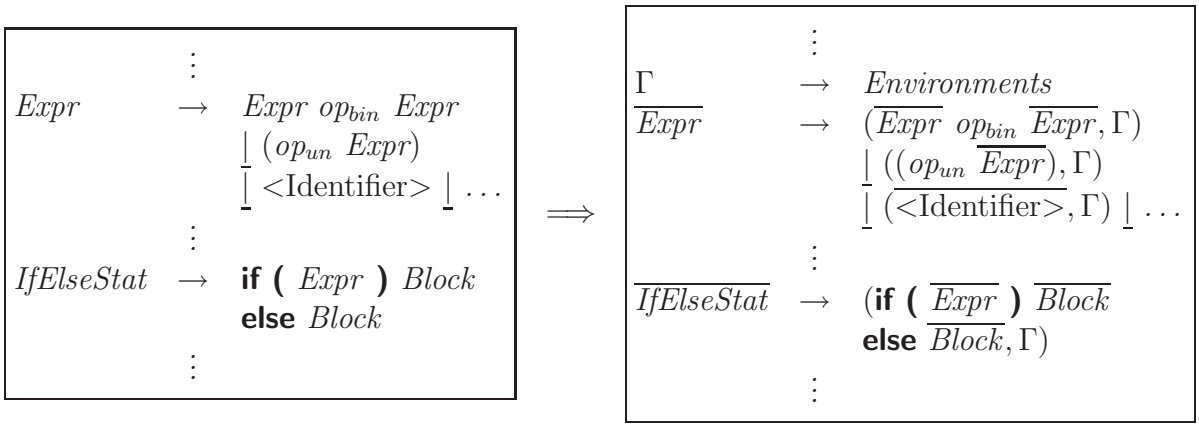


Abbildung 4.6: Erweiterung der ursprünglichen GANILA-Syntax (links) um statische Information (rechts).

oder Berechnungen auf dem AST ausgeführt werden. Die in den konkreten Umgebungen enthaltenen Informationen werden unmittelbar zur Generierung von JAVA-Code genutzt. Dabei benötigen wir, wie bereits erläutert, für Ausdrücke die Umgebungen ρ und ψ. Für Anweisungen werden die Umgebungen ρ, δ, γ sowie die Werte n und pp gebraucht.

Damit wir in den Funktionen keine graphische Notation von dekorierten ASTs angeben müssen, führen wir eine abkürzende textuelle Notation ein, deren Syntax in Abbildung 4.6 andeutungsweise präsentiert wird. Die Übersetzungsfunktionen benötigen in unserer Darstellung für die Übersetzung eines Programmkonstrukts folglich neben der Syntax auch die in der statischen Analyse berechneten Informationen. Das heißt wir übergeben den Funktionen für eine konkrete Anweisung *stat* als Argument ein Tupel $(stat, pp_{stat}, n_{stat}, \rho_{stat}, \delta_{stat}, \gamma_{stat})$ und kürzen dieses mit $\overline{stat}$ ab. Gleiches gilt für Ausdrücke, allerdings mit dem Unterschied, daß neben der Syntax nur zwei konkrete Umgebungen existieren: $\overline{expr} = (expr, \rho_{expr}, \psi_{expr})$. Betrachten wir als Beispiel die Übersetzungsfunktion für einen Ausdruck der Form $expr = expr_1 + expr_2$, dann gilt für die Definition der Funktion:

$$\boxed{\mathsf{gaja}_E \mid \overline{expr}} = \boxed{\mathsf{gaja}_E \mid (\overline{expr_1} + \overline{expr_2})\ \ \rho_{expr}\ \ \psi_{expr}}$$

Wir kürzen die Schreibweise in unseren Schemata weiter ab zu:

$$\boxed{\mathsf{gaja}_E \mid \overline{expr}} = \boxed{\mathsf{gaja}_E \mid (\overline{expr_1} + \overline{expr_2})\ \ \rho\ \ \psi}$$

Umgebungsvariablen ohne Index beziehen sich also immer auf das gerade zu übersetzende Programmkonstrukt.

4.4.1 Ausdrücke

Als erstes erläutern wir die Übersetzung von GANILA-Ausdrücken mit Hilfe der Codeerzeugungsfunktion gaja_E, die als Argumente einen Ausdrucksbaum sowie die dort abgespeicherten Informationen aus der statischen Analyse erhält. gaja_E zerlegt den Ausdruck

rekursiv und kombiniert die für die Teilausdrücke jeweils erzeugten JAVA-Codefragmente zu einem neuen JAVA-Ausdruck.

Wie bereits mehrfach erwähnt, werden in unserem Framework Variablen primitiven Typs in Objekte verpackt. Aus diesem Grund muß der Übersetzer alle in einem Ausdruck auftretenden Variablen primitiven Typs durch Variablen des Typs der entsprechenden Wrapperklasse ersetzen und sicherstellen, daß die Operanden eingebauter Operatoren R-Werte darstellen. Falls bestimmte Variablen während der Analyse für die Reifikation von PPs an neue Variablennamen gebunden worden sind, sind im generierten Code die neuen Variablennamen zu verwenden. Bevor wir die Übersetzung von Ausdrücken anhand einer Auswahl der wichtigsten Operatoren erläutern, führen wir eine neue Hilfsfunktion ein:

Hilfsfunktion: gaja$_{type}$ Eingabe für diese sehr einfache Funktion ist ein Typbezeichner. Handelt es sich um einen Referenztyp, dann wird eine Stringrepräsentation genau dieses Bezeichners ausgegeben. Wird der Bezeichner eines primitiven Datentyps übergeben, dann erzeugt GAJA den Typ der Wrapperklasse, also z. B. für den primitiven Typ `int`: gaja$_{type}$ | `int` $\Rightarrow$ Gint.

gaja$_{type}$ | *type* $\Rightarrow$

Fall 1: *type* $\in$ *ReferenceType* → *type*

Fall 2: *type* $\in$ *PrimitiveType* → G*type*

Variablen

Bezeichner für Variablen sind in Abhängigkeit vom Kontext, in dem sie auftreten, zu übersetzen. Dem umgebenden Ausdruck sind daher L- oder R-Werte dieser Variablen bereitzustellen. Eventuell sind Klone zu bilden. Enthält die Bindung einer Variablen *var* neue Variablennamen, d. h. $\rho(var) = ((type, pos), newVar)$ mit $newVar \neq \varepsilon$, dann ist der alte Bezeichner *var* durch die neue Variablenbezeichnung *newVar* zu ersetzen. Zu beachten ist, daß aus $pos =$ `global` direkt die Gleichheit $newVar = \varepsilon$ folgt, aber nicht umgekehrt. Es gibt auch lokal definierte Variablen, für die $newVar = \varepsilon$ gilt, beispielsweise wenn diese im Initialisierungsbereich einer **for**-Schleife definiert und angewandt werden.

gaja$_E$ | $(var)\ \rho\ \psi$ mit $\psi = (type, label)$, $\rho(var) = ((type, pos), newVar)$

$\Downarrow$

Fall 1: $newVar = \varepsilon$

Fall 1.1: $type \in ReferenceType \quad \vee \quad (type \in PrimitiveType \quad \wedge \quad label =$ `lVal`$)$

var

Fall 1.2: $type \in PrimitiveType \quad \wedge \quad label = \texttt{lCloneVal}$

var.Clone()

Fall 1.3: $type \in PrimitiveType \quad \wedge \quad label = \texttt{rVal}$

var.getValue()

Fall 2: $newVar \neq \varepsilon$

Fall 2.1: $type \in ReferenceType \quad \vee \quad (type \in PrimitiveType \quad \wedge \quad label = \texttt{lVal})$

$newVar$

Fall 2.2: $type \in PrimitiveType \quad \wedge \quad label = \texttt{lCloneVal}$

$newVar$.Clone()

Fall 2.3: $type \in PrimitiveType \quad \wedge \quad label = \texttt{rVal}$

$newVar$.getValue()

Geklonte L-Werte sind beispielsweise nötig, wenn eine Variable als Argument im Aufruf einer Methode der zu reifizierenden Klasse steht. Betrachten wir als Beispiel einen Methodenaufruf m(x); und nehmen weiterhin an, daß x eine global definierte Integervariable ist. Da die Variable in der Übersetzung gewrappt wird, und die Methode dort den Wrappertyp als Argumenttyp verlangt, würden wir ohne Klonen die Objektreferenz der Variablen direkt übergeben. Im Eingabeprogramm aber wird bei primitiven Typen aufgrund der JAVA-Semantik immer der Wert der Variablen übergeben. Der entsprechende formale Parameter im Rumpf der Methode würde dann auf dasselbe Objekt wie die globale Variable x zeigen, was nicht mehr mit der ursprünglichen Programmsemantik übereinstimmen würde.

Literale

Literale Werte haben meist einen primitiven Typ. Eine Ausnahme bilden in JAVA (und damit auch in GANILA) Konstanten vom Referenztyp String. Diese und R-Werte werden unverändert übersetzt. L-Werte von Literalen primitiven Typs müssen in ein neues Objekt gewrappt werden.

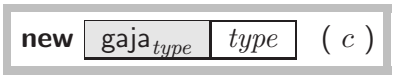
gaja_E	$(c)\ \rho\ \psi$

mit $\psi = (type, label)$

$\Downarrow$

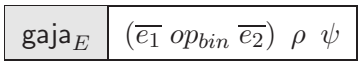
Fall 1: $type \in ReferenceType \quad \vee \quad label = \texttt{rVal}$

Fall 2: $type \in PrimitiveType \quad \wedge \quad label \in \{\texttt{lVal}, \texttt{lCloneVal}\}$



Binäre Operatoren

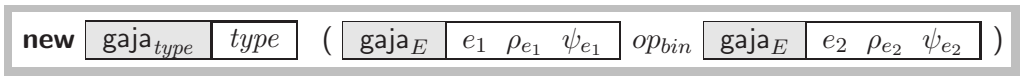
Wir geben ein allgemeines Übersetzungsschema für eine Reihe binärer Operatoren an. Ist der Ergebnistyp primitiv und wird im Kontext ein L-Wert erwartet, so erzeugen wir ein neues Wrapperobjekt. Zu beachten ist, daß der Operator **+** neben seiner arithmetischen Bedeutung in GANILA bzw. JAVA auch für eine Stringkonkatenation verwendet werden kann. In einem solchen Fall ist der Ergebnistyp ein Referenztyp und diese Operation wird wie in Fall 1 übersetzt.



mit $\psi = (type, label)$
$op_{bin} \in \{\texttt{+}, -, \texttt{*}, \texttt{/}, \texttt{\%}, \textsf{instanceof}, \texttt{!=}, \texttt{==}, \dots\}$

Fall 1: $type \in ReferenceType \quad \vee \quad (type \in PrimitiveType \quad \wedge \quad label = \texttt{rVal})$

Fall 2: $type \in PrimitiveType \quad \wedge \quad label \in \{\texttt{lVal}, \texttt{lCloneVal}\}$

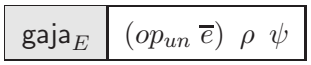


Die Typumgebung ρ bleibt für Ausdrücke und ihre jeweiligen Teilausdrücke unverändert, da weder eine Variablendeklaration innerhalb von Ausdrücken stattfinden kann[4], noch ein Aufruf einer Dispatchmethode erzeugt wird. Für binäre Ausdrücke gilt also: $\rho = \rho_{e_1} = \rho_{e_2}$.

[4] In JAVA kann über den **new**-Operator in einem Ausdruck eine neue *anonyme* Klasse implementiert werden. Die in dieser Klasse deklarierten Felder und Methoden sind im umgebenden Ausdruck allerdings nicht gültig.

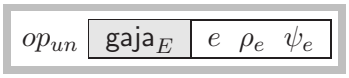
Unäre Operatoren

Die Übersetzung unärer Operatoren erfolgt analog. Diese sind nur auf primitiven Operandentypen definiert und liefern dementsprechend einen solchen Typ zurück.

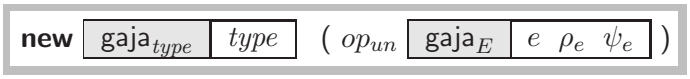


mit $\psi = (type, label)$
$type \in PrimitiveType$
$op_{un} \in \{+, -, !, \tilde{}\}$

Fall 1: $label =$ `rVal`

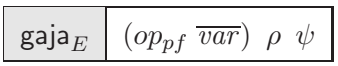


Fall 2: $label \in \{$`lVal`, `lCloneVal`$\}$



Inkrement- und Dekrementoperatoren

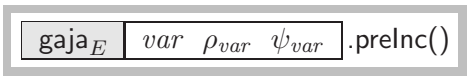
Inkrement- und Dekrementoperatoren in Präfix-Form erhöhen bzw. erniedrigen den Wert ihres Operanden um den Wert 1 und liefern den neuen Wert zurück. Bei dem Operanden muß es sich um eine Variable oder ein Feldelement mit numerischem Typ handeln; entsprechend ist auch der Ergebnistyp der Operatoren numerisch. Diese Operatoren können im übersetzten Code nicht mehr verwendet werden, weil der numerische Wert im Wrapperobjekt enthalten ist und nur indirekt geändert werden kann. Daher wurde ihre Funktionalität durch Methoden der Wrapperklassen implementiert, die im erzeugten Code wie folgt aufgerufen werden:



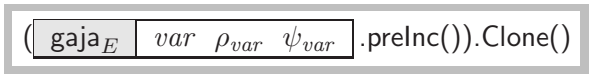
mit $\psi = (type, label)$
$type \in NumericType$
$op_{pf} \in \{++, --\}$

Fall 1: $op_{pf} = ++$

Fall 1.1: $label =$ `lVal`



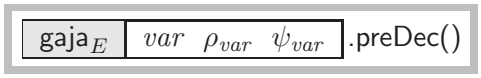
Fall 1.2: $label = \texttt{lCloneVal}$



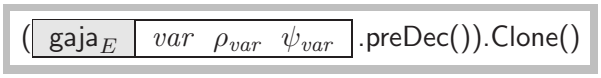
Fall 1.3: $label = \texttt{rVal}$

Fall 2: $op_{pf} = --$

Fall 2.1: $label = \texttt{lVal}$



Fall 2.2: $label = \texttt{lCloneVal}$



Fall 2.3: $label = \texttt{rVal}$

Analog zur Übersetzung von Variablen in Ausdrücken (vgl. S. 65 f.), sind dem umgebenden Ausdruck L- bzw. R-Werte als Ergebnis bereitzustellen. Das folgende Programmcodefragment zeigt die (vereinfachte) Implementierung der Wrapperklasse Gint für Integervariablen:

```
public class Gint extends GIntegralType {
  private int val;
  public Gint(int i) { val = i; }
  ...
  public int getValue() { return val; }
  public void setValue(int i) { val = i; }
  ...
  public void preDec() { --val; }
  public void preInc() { ++val; }
  ...
  public Gint Clone() { return new Gint(val); }
}
```

Die Übersetzung der Inkrement- und Dekrementoperatoren, die nach dem Operanden stehen und den alten Wert zurückliefern, erfolgt in ähnlicher Weise.

Zuweisungsausdrücke

Zuweisungen sind in der Programmiersprache JAVA Ausdrücke und werden daher mit der Funktion gaja_E übersetzt. Dazu greifen wir den einfachsten Fall heraus: die Zuweisung ohne weitere Operation (**=**). Zuweisungen mit Operationen, z. B. **+=**, **%=**, ..., sind analog zu übersetzen. Das Übersetzungsschema hat einige Ähnlichkeiten mit dem Schema für Inkrement- und Dekrementoperatoren. Es muß aber auch Referenztypen berücksichtigen.

$$\boxed{\mathsf{gaja}_E \mid (var = \overline{e})\ \ \rho\ \ \psi} \qquad \begin{array}{ll} \text{mit} & \psi = (type, label) \\ & \rho(var) = ((type, pos), newVar) \end{array}$$

$$\Downarrow$$

Fall 1: $type \in ReferenceType$

Fall 1.1: $newVar = \varepsilon$

$$var = \boxed{\mathsf{gaja}_E \mid e\ \ \rho_e\ \ \psi_e}$$

Fall 1.2: $newVar \neq \varepsilon$

$$newVar = \boxed{\mathsf{gaja}_E \mid e\ \ \rho_e\ \ \psi_e}$$

Fall 2: $type \in PrimitiveType \quad \wedge \quad newVar = \varepsilon$

Fall 2.1: $label =$ `lVal`

$$var.\mathsf{setValue}\ (\ \boxed{\mathsf{gaja}_E \mid e\ \ \rho_e\ \ \psi_e}\ )$$

Fall 2.2: $label =$ `lCloneVal`

$$(var.\mathsf{setValue}\ (\ \boxed{\mathsf{gaja}_E \mid e\ \ \rho_e\ \ \psi_e}\ )).\mathsf{Clone}()$$

Fall 2.3: $label =$ `rVal`

$$(var.\mathsf{setValue}\ (\ \boxed{\mathsf{gaja}_E \mid e\ \ \rho_e\ \ \psi_e}\ )).\mathsf{getValue}()$$

Fall 3: $type \in PrimitiveType \quad \wedge \quad newVar \neq \varepsilon$

Fall 3.1: $label =$ `lVal`

$newVar$.setValue (gaja$_E$ | e ρ_e ψ_e)

Fall 3.2: $label =$ `lCloneVal`

($newVar$.setValue (gaja$_E$ | e ρ_e ψ_e)).Clone()

Fall 3.3: $label =$ `rVal`

($newVar$.setValue (gaja$_E$ | e ρ_e ψ_e)).getValue()

4.4.2 Ausdrucksanweisungen

Unser erstes Beispiel für die Übersetzung von Anweisungen sind die sog. Ausdrucksanweisungen, d. h. Ausdrücke mit nachfolgendem Semikolon. Beispiele für Ausdrucksanweisungen sind Zuweisungen und Methodenaufrufe. In unserem Ansatz der Reifikation von Programmpunkten stellen Ausdrucksanweisungen Programmpunkte dar. GAJA erzeugt hier anstelle der Anweisung einen Aufruf zu einer neu generierten Dispatchmethode, in deren Rumpf die ursprüngliche Anweisung ausgeführt wird. Es ist also an zwei Stellen Programmcode zu generieren: für den Methodenaufruf und für die Definition der Dispatchmethode des betrachteten Programmpunkts.

gaja$_M$ | ($\overline{e}$;) pp n ρ δ γ mit $\delta(n)(pp) = (argVec, fpVec)$

$\Downarrow$

dispatch (pp, $argVec$) ;

public void dispatch_pp ($fpVec$) {
    gaja$_E$ | e ρ_e ψ_e ;
}

Hier ist auch die Verwendung der Umgebung δ zu erkennen, aus der die Argumente für den Dispatchmethodenaufruf bzw. die formalen Parameter für die Definition der Dispatchmethode erzeugt werden.

4.4.3 Kontrollflußanweisungen

Zuweisungen ändern den Zustand eines Programms, aber nicht dessen Kontrollfluß. Im folgenden betrachten wir zwei „komplexere“ Kontrollkonstrukte, die sich zur Laufzeit auf den Kontrollfluß des zu übersetzenden Programms auswirken können: die bedingte Anweisung sowie die **return**-Anweisung mit Rückgabewert. GAJA erzeugt für beide Konstrukte ebenfalls eigene Dispatchmethoden. Unser Hauptaugenmerk wird auf der Übersetzung der Verzweigungsanweisungen **return**, **break** und **continue** liegen, die alle im Rumpf komplexerer Kontrollflußanweisungen, wie etwa einer bedingten Anweisung oder einer Schleife, auftreten können. Dazu sind die bereits erwähnten Kontrollflußobjekte des Typs ControlFlow zu bilden und zur Laufzeit zu ihrem Zielblock zu propagieren.

Um diese Transformation von *intra*prozeduralem Kontrollfluß zu *inter*prozeduralem Kontrollfluß zu vereinfachen, definieren wir drei Hilfsfunktionen, die von den noch zu diskutierenden Übersetzungsfunktionen der beiden Kontrollflußanweisungen aufgerufen werden. Die Hilfsfunktionen erhalten die aktuelle Nummer des PP und die Kontrollflußumgebung γ als Argumente und generieren Code, der unmittelbar nach der Abarbeitung einer Dispatchmethode im Block des Dispatchmethodenaufrufs ausgeführt wird. Der generierte JAVA-Code überprüft das von einer Dispatchmethode zurückgelieferte Kontrollflußobjekt und entscheidet zur Laufzeit, ob es an eine umschließende Dispatchmethode weitergeleitet oder im aktuellen Methodenrumpf entsprechend seiner Semantik verarbeitet wird.

Hilfsfunktion: gaja$_{ret}$ Diese Funktion generiert Code für die Behandlung von Kontrollflußobjekten, die von **return**-Anweisungen mit und ohne Rückgabewert initiiert worden sind. Sie erzeugt keinen Code, wenn im Unterbaum des aktuellen PP keine **return**-Anweisung enthalten ist. Falls doch und ist der umgebende Block des aktuellen PP der Rumpf einer Methode, dann wird die dem PP entsprechende Anweisung ausgeführt und ein Wert zurückgeliefert, falls das Kontrollflußobjekt cfl einen Wert enthält. Kontrollflußobjekte enthalten Werte vom allgemeinen Typ Object, daher ist der Rückgabewert entsprechend seines über die Typinferenz ermittelten Typs zu konvertieren. Wir müssen in diesem Prozeß zur Generierungszeit unterscheiden, ob eine **return**-Anweisung immer auftritt, oder nur in bestimmten Fällen. Der Standardcompiler JAVAC führt diese Analyse ebenfalls durch und gibt eine Fehlermeldung aus, falls eine Methode mit Rückgabewert nicht unter allen Umständen einen Wert zurückliefert. Enthält diese Methode im Eingabeprogramm beispielsweise eine **return**-Anweisung, die im Block einer einseitigen bedingten Anweisung steht, dann kann zur späteren Laufzeit ein Kontrollflußobjekt mit Rückgabewert in Abhängigkeit von der Bedingung erzeugt werden oder nicht. Daher muß im erzeugten Methodenrumpf ein entsprechender Test durchgeführt werden, wie in Fall 2.2 des nachfolgenden Schemas zu sehen ist.

gaja$_{ret}$	$pp\ \gamma$

mit $\gamma(pp) = (envBlock, cflOps, (type, prob))$

$\Downarrow$

Fall 1: `return` $\notin$ *cflOps*

ε

Fall 2: `return` $\in$ *cflOps* $\wedge$ *envBlock* = `method`

Fall 2.1: *prob* = `s`

return $\begin{cases} (\ \boxed{\text{gaja}_{type} \mid type}\ )\ \text{cfl.value}\ ; & type \neq \texttt{void} \\ ; & type = \texttt{void} \end{cases}$

Fall 2.2: *prob* = `m`

if (cfl.isReturn) $\begin{cases} \textbf{return}\ (\ \boxed{\text{gaja}_{type} \mid type}\ )\ \text{cfl.value}\ ; & type \neq \texttt{void} \\ \textbf{return}\ ; & type = \texttt{void} \end{cases}$

Fall 3: `return` $\in$ *cflOps* $\wedge$ *envBlock* $\neq$ `method`

if (cfl.isReturn) **return** cfl ;

Ist der umgebende Block nicht der Rumpf einer im Eingabeprogramm definierten Methode (Fall 3), so wird das Kontrollflußobjekt weiter propagiert, bis es seinen Methodenrumpf (Zielblock) erreicht.

Hilfsfunktion: gaja$_{break}$ Die Hilfsfunktion gaja$_{break}$ erzeugt keinen Code, wenn im Unterbaum des aktuellen PP keine **break**-Anweisung enthalten ist. Falls doch, dann wird eine Abfrage generiert, welche die **break**-Anweisung genau dann zur Laufzeit ausführt, wenn der umgebende Block der Zielblock, d. h. eine Schleife oder ein **switch**-Konstrukt, der Anweisung ist. Die Markierung `loop` für Schleifen steht zusammenfassend für `for`-, `while`- und `do`-Schleifen.

$\boxed{\text{gaja}_{break} \mid pp\ \gamma}$ mit $\gamma(pp) = (envBlock, cflOps, (type, prob))$

$\Downarrow$

Fall 1: `break` $\notin$ *cflOps*

ε

Fall 2: `break` $\in$ *cflOps*

if (cfl.isBreak) $\begin{cases} \textbf{break}\ ; & envBlock \in \{\texttt{loop}, \texttt{switch}\} \\ \textbf{return}\ \text{cfl}\ ; & \text{sonst} \end{cases}$

Hilfsfunktion: gaja$_{cont}$ Unsere letzte Hilfsfunktion hat eine ähnliche Semantik wie gaja$_{break}$. Der Unterschied liegt im Zielblock der **continue**-Anweisung, der hier lediglich aus einem Schleifenkonstrukt bestehen kann. Entspricht also der umgebende Block einem Schleifenrumpf, dann erzeugt gaja$_{cont}$ Programmcode, der eine **continue**-Anweisung zur Laufzeit ausführt.

gaja$_{cont}$	$pp\ \gamma$

mit $\gamma(pp) = (envBlock, cflOps, (type, prob))$

$$\Downarrow$$

Fall 1: `continue` $\notin$ *cflOps*

ε

Fall 2: `continue` $\in$ *cflOps*

if (cfl.isContinue) $\begin{cases} \textbf{continue ;} & envBlock = \texttt{loop} \\ \textbf{return}\ \text{cfl ;} & \text{sonst} \end{cases}$

Bedingte Anweisungen

Mit Hilfe der drei Hilfsfunktionen wird die Beschreibung der Übersetzungsfunktionen für bedingte Anweisungen sowie für **return**-Anweisungen stark vereinfacht. Das unten definierte Übersetzungsschema für bedingte Anweisungen unterscheidet nur zwei Fälle: Ist in den beiden Blöcken der bedingten Anweisung keine einzige **return**-, **break**- oder **continue**-Anweisung enthalten, dann braucht GAJA keinen Code für das Kontrollflußhandling zu erzeugen. Fall 1 zeigt auf der linken Seite die Generierung des Dispatchmethodenaufrufs anstelle der ursprünglichen bedingten Anweisungen. Auf der rechten Seite ist der zu erzeugende Code der zugeordneten Dispatchmethode dargestellt, welcher drei Aufrufe der Übersetzungsfunktionen für die beiden Blöcke sowie für die Bedingung enthält.

In Fall 2 wird angenommen, daß innerhalb der bedingten Anweisung mindestens eine der o. g. Anweisungen enthalten ist. Die generierte Dispatchmethode (rechte Seite) deklariert zunächst eine Variable für ein Kontrollflußobjekt. Dieser Variablen wird evtl. mehrfach ein von den Übersetzungsfunktionen der beiden Blöcke zurückgeliefertes Kontrollflußobjekt zugewiesen. Die Übersetzungsfunktionen der beiden Blöcke erzeugen dabei Code für deren Behandlung. Am Ende des Dispatchmethodenrumpfes wird ein „leeres“ Kontrollflußobjekt zurückgeliefert, falls dies noch nicht geschehen ist[5]. Das zurückgelieferte Kontrollflußobjekt wird schließlich von dem Programmcode behandelt, welcher von den drei Hilfsfunktionen erzeugt wurde.

[5] Diese Anweisung wird möglicherweise nie erreicht.

gaja_M	(**if** $(\overline{e})$ $\overline{b_1}$ **else** $\overline{b_2}$) pp n ρ δ γ

mit $\delta(n)(pp) = (argVec, fpVec)$
$\gamma(pp) = (envBlock, cflOps, (type, prob))$

$$\Downarrow$$

Fall 1: $cflOps = \emptyset$

dispatch (pp, $argVec$) ;

public void dispatch_pp ($fpVec$) {
  if (gaja_E | e ρ_e ψ_e) {
    gaja_M | b_1 pp_{b_1} n_{b_1} ρ_{b_1} δ_{b_1} γ_{b_1}
  } **else** {
    gaja_M | b_2 pp_{b_2} n_{b_2} ρ_{b_2} δ_{b_2} γ_{b_2}
  }
}

Fall 2: $cflOps \neq \emptyset$

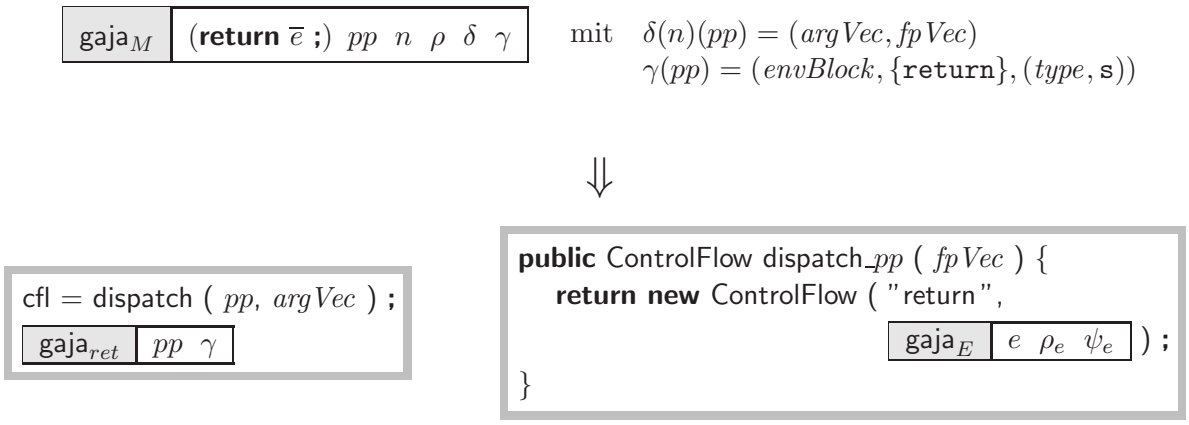
cfl = dispatch (pp, $argVec$) ;
gaja_{ret} | pp γ
gaja_{break} | pp γ
gaja_{cont} | pp γ

public ControlFlow dispatch_pp ($fpVec$) {
  ControlFlow cfl ;
  if (gaja_E | e ρ_e ψ_e) {
    gaja_M | b_1 pp_{b_1} n_{b_1} ρ_{b_1} δ_{b_1} γ_{b_1}
  } **else** {
    gaja_M | b_2 pp_{b_2} n_{b_2} ρ_{b_2} δ_{b_2} γ_{b_2}
  }
  return new ControlFlow() ;
}

Die Übersetzungsfunktionen für die drei Teilkomponenten der bedingten Anweisung erhalten als Argumente die erweiterte Syntax zusammen mit ihren Umgebungen. Üblicherweise unterscheiden sich für einen PP fast alle Umgebungsvariablen. In diesem Schema gilt nur, daß $\rho = \rho_e$ und $\gamma = \gamma_{b_1} = \gamma_{b_2}$. Die erste Gleichheit resultiert aus dem Umstand, daß Ausdrücke keine eigenen PPs darstellen und keine eigenen Dispatchmethoden erhalten. Daher werden der aktuellen Typumgebung auch keine Bindungen an neue Variablenbezeichner für übergebene Variablen hinzugefügt. Weiterhin sind neue Variablendefinitionen innerhalb der Blöcke der bedingten Anweisung im Ausdruck nicht gültig. Die zweite Gleichheit ist für dieses Schema spezifisch: Es handelt sich bei der bedingten Anweisung weder um eine Schleife, noch um eine Methode oder ein **switch**-Konstrukt, so daß sich der Umgebungskontext hinsichtlich möglicher Zielblöcke für Kontrollflußobjekte nicht verändert.

return-Anweisungen mit Rückgabewert

Als letztes Beispiel für ein Codeerzeugungsschema betrachten wir die Übersetzung von **return**-Anweisungen mit einem Rückgabewert. Dieses Schema ist wesentlich einfacher als das zuletzt betrachtete. Zur Generierungszeit ist der Kontrollfluß für dieses Konstrukt vollständig klar. Die erzeugte Dispatchmethode (rechte Seite des Schemas) für diesen PP liefert ein neues Kontrollflußobjekt zurück, welches eine Markierung für die Anweisung **return** enthält, sowie den entsprechenden Wert des in der Anweisung enthaltenen Ausdrucks. Das Kontrollflußobjekt wird zur Laufzeit von Dispatchmethode zu Dispatchmethode weitergereicht, bis der Rumpf einer (ursprünglich im Eingabeprogramm enthaltenen) Methode erreicht wird. Dort, im Zielblock dieser Anweisung, wird dann eine **return**-Anweisung ausgeführt, wie in der Beschreibung der Hilfsfunktion gaja_{ret} bereits erläutert wurde.



5 Die GANIMAL-Laufzeitumgebung

In den vorangegangenen Kapiteln wurden Syntax und Semantik der Animationsbeschreibungssprache Ganila angegeben und erläutert, welche Module der Compiler GaJa aus einem gegebenen Ganila-Programm erzeugt. Diese fünf Module, insbesondere das Algorithmenmodul, bilden im Zusammenspiel mit einer Laufzeitumgebung interaktive, multimediale Animationen des Eingabeprogramms. Hierbei repräsentieren Sichten graphisch und auditiv die verschiedenen Berechnungszustände. Die Ganimal-Laufzeitumgebung enthält eine Reihe von vordefinierten Sichten, einschließlich einer Graphensicht, einer Quellcodesicht, einer Sicht für Dokumentationen in HTML und einer „Aura“ für akustisches Feedback oder gesprochene Erklärungen. Benutzerdefinierte Sichten können in einfacher Weise durch die Implementierung von Unterklassen geschaffen werden. Die Laufzeitumgebung ermöglicht dem Benutzer weitreichende Interaktionsmöglichkeiten, indem sie eine graphische Benutzeroberfläche zur Kontrolle und Modifikation einer Animation zur Verfügung stellt.

In diesem Kapitel diskutieren wir die Eigenschaften, den Aufbau und einige interessante Implementierungsaspekte der Laufzeitumgebung des Ganimal-Frameworks. Zuerst beleuchten wir globale Systemzusammenhänge anhand des MVC-Entwurfsmusters. Anschließend gehen wir auf die einzelnen Systemteile ein, die jeweils den unterschiedlichen Komponenten des Musters entsprechen. Eine detaillierte Beschreibung der Ganimal-Laufzeitumgebung findet man in der Diplomarbeit [Wel01].

5.1 Realisierung einer MVC-Architektur

Die Architektur der Ganimal-Laufzeitumgebung basiert auf dem sogenannten MVC-Entwurfsmuster (Model/View/Controller-Entwurfsmuster). MVC unterteilt ein System in drei Hauptkomponenten [GHJV95]:

Model Das Modell ist das Anwendungsobjekt und stellt den funktionalen Kern der Anwendung dar. Es kapselt die Daten und enthält Methoden, die auf diesen Daten arbeiten.

View Eine Sicht präsentiert das Modell auf dem Bildschirm und informiert den Benutzer über dessen Zustand. Zu einem Modell können mehrere, verschiedene Sichten existieren.

Controller Darunter versteht man die Programmeinheit, die die Interaktion mit dem Anwender steuert, z. B. die Verarbeitung von Tastatur- und Mausereignissen.

Ein Hauptvorteil von MVC liegt darin, daß dieses Muster Modell und Sichten strikt voneinander trennt. Dazu wird ein Registrierungs- und Benachrichtigungsprotokoll über die Kontrolle etabliert. Auf der einen Seite muß eine Sicht jederzeit sicherstellen, daß ihr Inhalt dem Modellzustand entspricht. Auf der anderen Seite hat das Modell jede registrierte Sicht von Änderungen seines Zustands zu unterrichten. Durch dieses einfache Verfahren können viele Sichten auf ein einziges Modell implementiert werden, ohne das Modell selbst zu modifizieren. Darüber hinaus können Sichten unabhängig voneinander implementiert und sogar zur Laufzeit ausgetauscht werden, was aus der Realisierung des Entwurfsmusters *Observer* resultiert. Allgemein ist MVC eine Komposition mehrerer, allgemeinerer Muster. Neben dem Muster Observer spielen auch die Muster *Composite* und *Strategy* eine wichtige Rolle, wie in dem Lehrbuch von Gamma et al. [GHJV95] beschrieben wird. Wir werden den Gebrauch dieser beiden Muster in Abschnitt 5.3 über die Visualisierungskontrolle und in 5.4 über die Sichten deutlich machen.

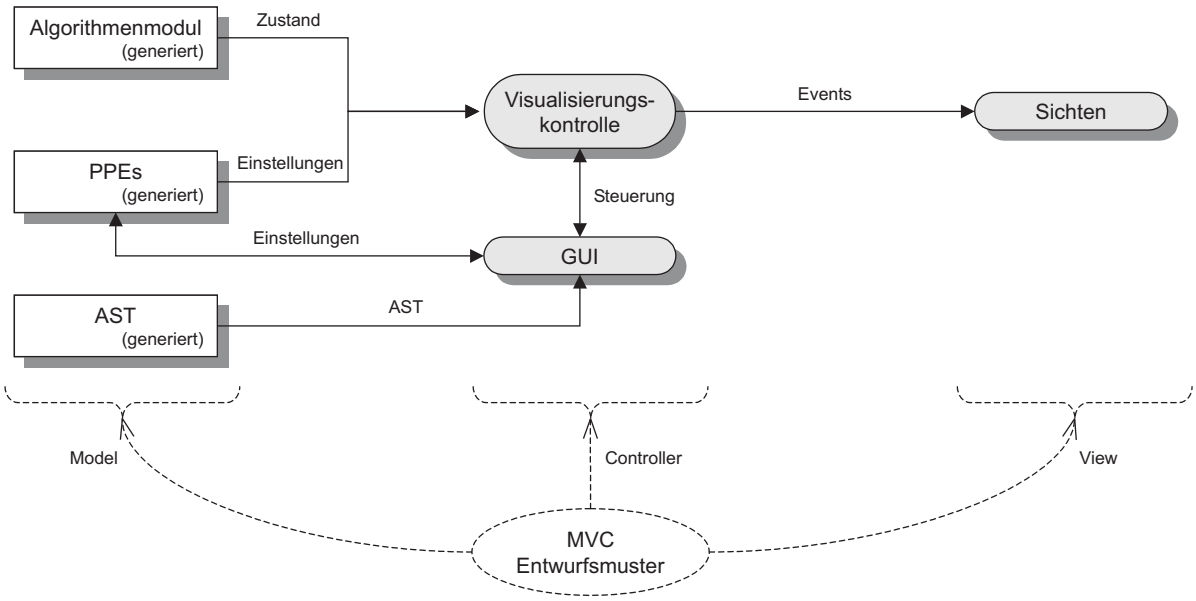


Abbildung 5.1: Die GANIMAL-Laufzeitumgebung[1].

Abbildung 5.1 zeigt eine erste Grobstruktur der Laufzeitumgebung von GANIMAL und weist auf die einzelnen Komponenten des MVC-Musters hin. Drei von GAJA generierte Klassen, das Algorithmenmodul, die (initialen) PPEs sowie der AST, entsprechen der Model-Komponente des MVC-Musters. Sie wurden im letzten Kapitel ausführlich beschrieben, daher verzichten wir im folgenden auf deren weitere Diskussion und konzentrieren unsere Beschreibung auf die verbleibenden MVC-Komponenten: Unsere graphischen Sichten auf den zugrundeliegenden Algorithmus korrespondieren, wie die Namensgebung bereits vermuten läßt, zu den View-Komponenten. Sie erhalten Informationen

[1] Im Verlauf dieses Kapitels werden wir die einzelnen Komponenten dieses Diagramms schrittweise verfeinern und als UML-Diagramme darstellen: die Visualisierungskontrolle mit zugehöriger GUI in Abbildung 5.4, den Eventmechanismus zur Kommunikation zwischen Controller- und View-Komponente in Abbildung 5.5 sowie die Architektur der Sichten in Abbildung 5.6.

über den Algorithmus und dessen Datenstrukturen durch den Empfang von Events. Benutzerinteraktionen hinsichtlich der Animationssteuerung und Modifikation beliebiger PPEs führt die Laufzeitumgebung mit Hilfe der graphischen Benutzerschnittstelle und einer Reihe von Kontrollobjekten aus. Diese Kontrollobjekte bilden gemeinsam die Visualisierungskontrolle der Laufzeitumgebung. Die GUI stellt die graphische Repräsentation eines abstrakten Syntaxbaums, der aus dem zugrundeliegenden Eingabeprogramm erzeugt wurde, als Benutzerschnittstelle zur Verfügung. Somit könnte die GUI im Sinne des MVC-Musters ebenfalls als eine Sicht auf das Modell interpretiert werden. Aufgrund ihrer Sonderstellung — sie modifiziert PPEs unmittelbar und dient ausschließlich als graphische Schnittstelle zu den Kontrollobjekten — ordnen wir sie zusammen mit der Visualisierungskontrolle der Controller-Komponente zu.

5.2 Graphische Benutzerschnittstelle

Die GANIMAL-Laufzeitumgebung stellt dem Benutzer eine graphische Oberfläche zur Verfügung. Sie bietet zwei Hauptfunktionalitäten: die Steuerung des Animationsablaufs und das Verändern der PPEs für bestimmte Programmpunkte vor und während der Animation.

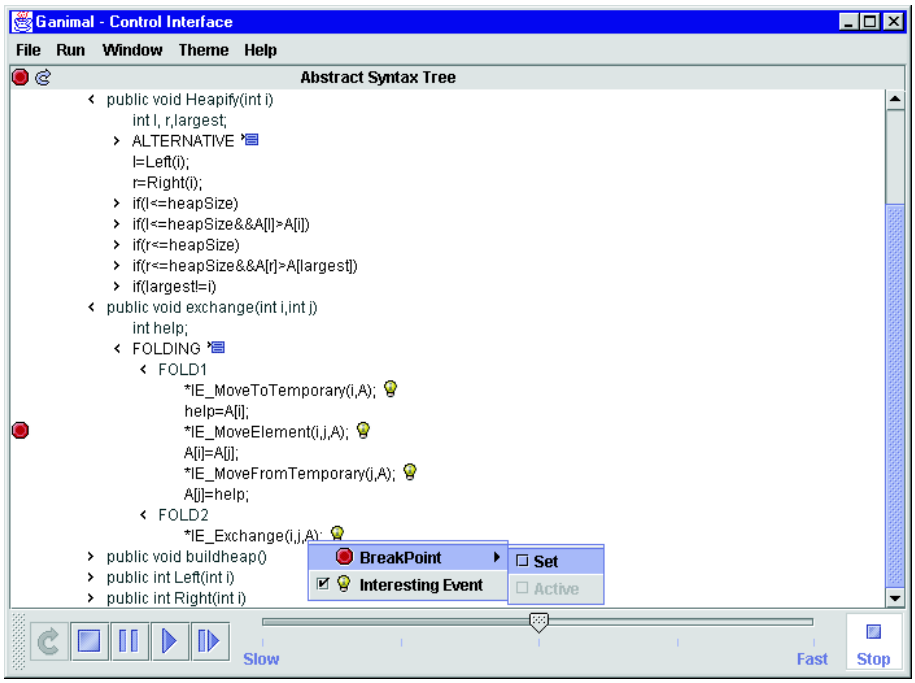


Abbildung 5.2: Die graphische Benutzerschnittstelle des GANIMAL-Frameworks.

Abbildung 5.2 zeigt eine Bildschirmaufnahme der GUI. Sie ist in drei Bereiche aufgeteilt: Der obere Bereich besteht aus einem Menü, über das man eine in GANILA erstellte Algorithmenanimation laden kann. Dies geschieht über den Aufruf der auf Seite 52

vorgestellten, generierten Konfigurationsdatei. Weitere Menüpunkte ermöglichen eine individuelle Anordnung der verschiedenen Sichten, eine Anpassung des graphischen Erscheinungsbildes der Fenster, den Aufruf einer Hilfe und die Steuerung der Animation. Eine redundante Steuerungsmöglichkeit besteht in einer Gruppe von Schaltflächen, die im unteren Bereich der GUI erkennbar ist. Der mittlere Bereich unterstützt den Benutzer bei der Veränderung der PPEs. Die Modifikation der PPEs sowie die Steuerung einer Animation werden in 5.2.1 und 5.2.2 diskutiert.

Es ist jedoch nicht immer erwünscht, daß alle Bereiche sichtbar sind. Daher lassen sich einzelne Bereiche wegblenden und somit auf dem Bildschirm Freiflächen zur Darstellung der Sichten schaffen. Diese Eigenschaft ist insbesondere dann nützlich, wenn der Visualisierer keine Modifikation der PPEs zulassen möchte, z. B. in kleineren JAVA-Applets.

5.2.1 Steuerung einer Animation

Zur Steuerung der Animation stehen mehrere Schaltflächen und ein Geschwindigkeitsregler in einer Werkzeugleiste zur Verfügung, die sich entweder als separates Hilfsmittelfenster auf dem Bildschirm an beliebiger Stelle plazieren oder im Fenster der GUI fest verankern läßt. Durch das Betätigen der Schaltflächen kann der Benutzer die Animation anhalten und zum Anfang zurückspringen, pausieren, oder die Animation automatisch mit einer bestimmbaren Geschwindigkeit ablaufen lassen bzw. schrittweise ausführen[2]. In der Werkzeugleiste ist (ganz rechts) ein Symbolfeld vorhanden, das den aktuellen Steuerungszustand sowie den Animationsmodus anzeigt.

Die GUI interagiert hierzu mit der Visualisierungskontrolle. Beide Komponenten übermitteln sich wechselseitig Veränderungen im Steuerungszustand. Das System versendet direkte Benutzerinteraktionen als Nachricht zu einem Kontrollobjekt, welches beispielsweise den Thread des Algorithmenmoduls stoppen läßt. Weiterhin muß das verantwortliche Kontrollobjekt einen auftretenden Haltepunkt nicht nur im Algorithmenmodul ausführen, sondern auch die GUI unmittelbar davon in Kenntnis setzen. Abbildung 5.3 stellt die Position der GUI-Komponente innerhalb der GANIMAL-Laufzeitumgebung dar.

5.2.2 Modifikation der Programmpunkteinstellungen

Die GUI bietet im mittleren Bereich von Abbildung 5.2 eine Schnittstelle, die einem Benutzer alle aktuellen PPEs anzeigt. Der Compiler GAJA erzeugt aus einem GANILA-Eingabeprogramm zwei Module, die von der GUI zur Realisierung dieser Schnittstelle verwendet werden: die (initialen) PPEs sowie eine Repräsentation des AST des Eingabeprogramms. Der Syntaxbaum wird im Fenster der GUI dargestellt. Eine Teilmenge

[2] In der Bildschirmaufnahme ist ein weiterer inaktiver Button im unteren Bereich ganz links zu erkennen. Eine Nachfolgerversion des Frameworks soll die Angabe von Rücksetzpunkten ermöglichen, die in GANILA mit dem Token ***SAVE** markiert werden sollen. Wenn die Ausführung eines Algorithmus einen solchen Punkt erreicht, kopiert das System den aktuellen Zustand und speichert ihn in einer History. Rücksetzpunkte sind ein Mittel für die Rückwärtsausführung eines Algorithmus bzw. für die Wiederholung eines Algorithmus von einem vorhergehenden Punkt aus, unter Umständen mit geänderten PPEs.

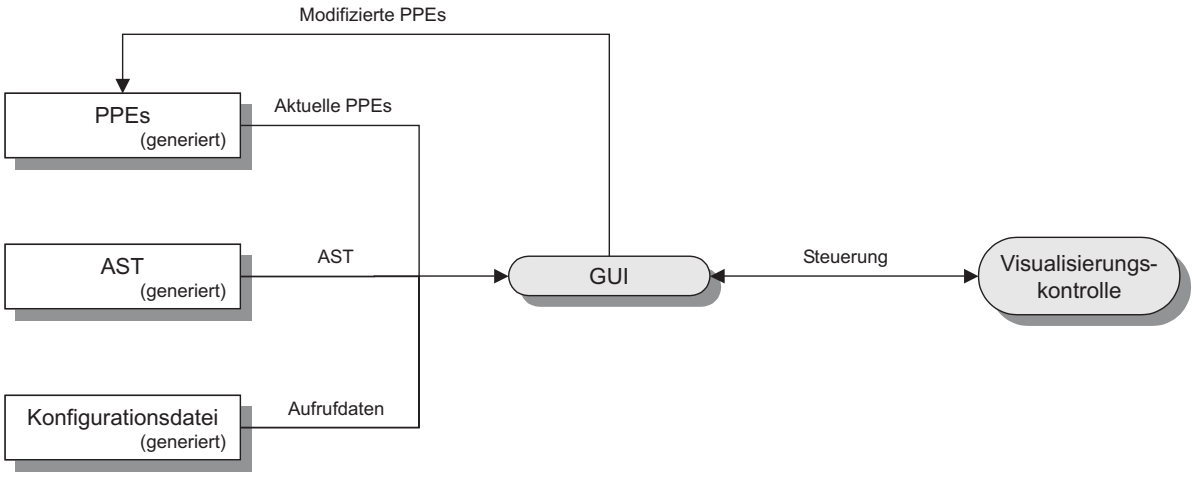


Abbildung 5.3: Position der GUI in der GANIMAL-Laufzeitumgebung.

der Knoten entspricht den Programmpunkten des Eingabeprogramms, und der Benutzer kann an diesen Knoten jede einzelne PPE vor dem Start oder zur Laufzeit der Animation verändern (vgl. auch Abschnitt 3.2.1 auf S. 31).

Ein sog. Explorerlayout der Baumdarstellung bietet hierbei eine Reihe von Vorteilen: Die Strukturierung des Programmcodes ergibt sich automatisch aus der entsprechenden Hierarchiestufe, d. h. die Rümpfe von komplexeren Kontrollstrukturen wie von Schleifen oder bedingten Anweisungen werden eingerückt angezeigt. Alle Klassen und Methodenrümpfe werden ebenfalls eingerückt. Damit findet sich ein Benutzer im Eingabeprogramm leicht zurecht. Unsere Implementierung unterstützt zusätzlich die Möglichkeit, uninteressante Codefragmente durch einen Mausklick auf den Elternknoten auszublenden. Ferner sind alle Programmcodezeilen, die einen Programmpunkt repräsentieren, schwarz abgebildet. Alle sonstigen Programmcodezeilen, wie zum Beispiel Deklarationen oder Import-Anweisungen, sind dunkelgrau repräsentiert. Ist die Animation des Algorithmus im Ablauf, so markiert die GUI die aktuelle Codezeile in roter Textfarbe.

Knoten, die einen Programmpunkt und ein GANILA-Konstrukt darstellen, sind mit einem individuellen graphischen Icon versehen, das den Aktivierungszustand seiner Metainformationen (PPEs) anzeigt. Über einen Mausklick auf den Knotentext bzw. auf das Icon von GANILA-Konstrukten läßt sich ein Popup-Menü öffnen, mit dem sich die PPEs für den jeweiligen Programmpunkt ändern lassen. Dabei werden nur die PPEs angezeigt, bei denen die Möglichkeit einer Veränderung besteht. Mit Tabelle 5.1 ist eine Gegenüberstellung der von unserer Implementierung unterstützten PPEs gegeben. Alle PPEs werden vom Visualisierer über GANILA-Annotationen im Eingabeprogramm definiert und per Voreinstellung aktiviert. Einige davon können aber auch zur Laufzeit der Algorithmenanimation über die GUI modifiziert werden. Nach jeder Änderung aktualisiert die GUI die Werte der PPEs, welche durch den Compiler GAJA initial vorgegeben wurden (siehe Abb. 5.3). Dadurch kann die Visualisierungskontrolle immer auf die aktuellen Werte zugreifen.

PPEs	(de)aktivierbar	definierbar
Haltepunkte	•	•
Modi		
#Besuche	•	•[3]
Interesting Events	•	
Invarianten	•	•[3]
Parallele Ausführung	•	
Alternativen	•	
Falten	•	

Tabelle 5.1: Modifizierbarkeit der PPEs über die GUI.

5.3 Visualisierungskontrolle

Die Visualisierungskontrolle ist die zentrale Verbindungsstelle zwischen den Klassen des Modells und den verschiedenen graphischen Sichten auf den Eingabealgorithmus. Sie schickt den Sichten parametrisierte Instanzen von Interesting Events, die Informationen über den aktuellen Berechnungs- und Animationszustand beinhalten. Diese Zustände sind von den aktuell ausgeführten reifizierten Programmpunkten mit ihren jeweiligen Einstellungen (PPEs) abhängig und von Benutzerinteraktionen, die über die GUI vorgenommen werden können. Unsere Implementierung der Visualisierungskontrolle umfaßt mehrere Kontrollklassen, die unterschiedliche Aufgaben lösen. Zu diesen Aufgaben gehören die Steuerung der Animation und des Algorithmus zur Laufzeit, die Kommunikation mit den Sichten und die Verarbeitung der durch den Compiler GAJA übersetzten GANILA-Annotationen.

5.3.1 Steuerung des Algorithmus

Sobald der Benutzer eine Animation über die GUI in das Laufzeitsystem lädt, wird in einem ersten Schritt jede generierte Klasse instantiiert. Hierbei werden alle deklarierten Sichten über das Kontrollobjekt GAlgorithm registriert und in der Benachrichtigungsklasse GEventControl abgespeichert. Diese Klasse ist für das Versenden von Interesting Events verantwortlich. Abbildung 5.4 zeigt ein Klassendiagramm, das diese Zusammenhänge graphisch veranschaulicht. Nach dem Start der Animation werden die reifizierten Programmpunkte in der Reihenfolge ihres Auftretens in einem eigenen Thread abgearbeitet. Wie in Kapitel 3 ausführlich dargestellt, ruft das Algorithmenmodul dazu die generische Methode dispatch() auf, die per JAVA-Reflection die entsprechend generierte dispatch_*pp*()-Methode ausführt. Über die generische Methode dispatch() kann das Laufzeitsystem die Animation vor der Ausführung eines reifizierten Programmpunkts anhalten und überprüfen, ob

1. der Benutzer beispielsweise eine Schaltfläche in der GUI betätigt hat, oder ob
2. die PPEs eine Veränderung des Animationsablaufs vorschreiben.

[3]Geplant für eine zukünftige Version des Laufzeitsystems.

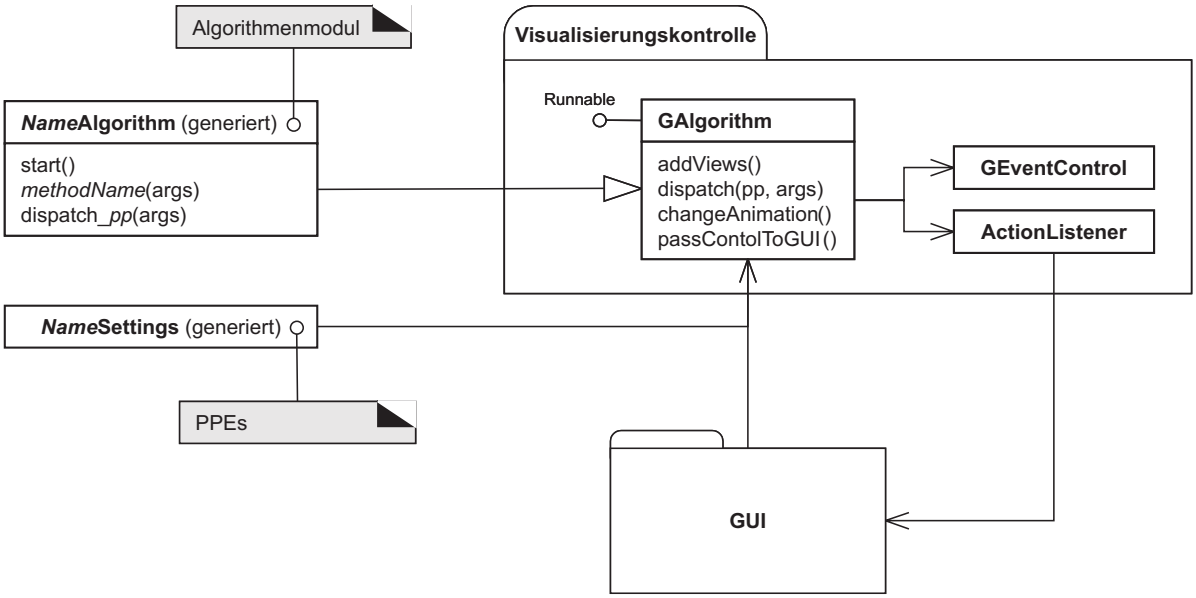


Abbildung 5.4: Kontrolle zur Algorithmensteuerung (UML-Notation).

Im ersten Fall ruft die generische dispatch()-Methode eine weitere Methode passControlToGUI() auf, die den Thread des Algorithmenmoduls in Abhängigkeit vom Steuerungszustand und vom gegebenen Geschwindigkeitsfaktor kontrolliert. Der Steuerungszustand wird von der GUI selbst verändert, wenn der Benutzer eine Schaltfläche drückt oder eine Menüauswahl vornimmt. Dazu führt die GUI die Methode changeAnimation() der Klasse GAlgorithm mit einer Benutzerinteraktion als Argument aus, also z. B. die schrittweise Ausführung oder das Anhalten des Algorithmus. Der Steuerungszustand kann ebenso von Konstrukten des Eingabeprogramms (z. B. von Haltepunkten) modifiziert werden. Das Laufzeitsystem muß einem Benutzer derartige Ereignisse mitteilen und in der GUI deutlich machen. Dazu schickt die Methode passControlToGUI() eine Nachricht an einen sog. ActionListener, der sie an die GUI weiterleitet. So können sich auch andere (evtl. zukünftige) Komponenten des Laufzeitsystems bei diesem ActionListener anmelden und die gleichen Nachrichten empfangen.

Der zweite Punkt ist Gegenstand von Abschnitt 5.3.2. Obwohl Haltepunkte und Animationsmodi ebenfalls PPEs darstellen, diskutieren wir sie hier. Der Grund liegt darin, daß sie zum einen den Steuerungszustand verändern und zum anderen mit jedem beliebigen Programmpunkt des Eingabeprogramms verknüpft sein können, was sie von allen anderen Arten von PPEs unterscheidet.

Haltepunkte

Die generische dispatch()-Methode überprüft die PPEs für jeden reifizierten Programmpunkt. Handelt es sich bei diesem um einen Haltepunkt und ist der Haltepunkt aktiviert, so wird die Ausführung des Algorithmus solange angehalten, bis in der GUI eine Schaltfläche zur weiteren Ausführung betätigt wird.

Animationsmodi

Das Algorithmenmodul speichert den aktuellen Modus (*RECORD* oder *PLAY*), in dem sich die Algorithmenanimation befindet. Es ändert den abgespeicherten Modus, sobald in der Ausführung ein Programmpunkt erreicht wird, dessen PPE einen Moduswechsel veranlaßt. Dazu wurden die Programmpunkte im Quellcode des Eingabeprogramms vom Visualisierer mit den GANILA-Konstrukten ***REPLAY** und ***RECORD** versehen. Befindet sich die Animation im *RECORD*-Modus, dann speichert das o. g. Kontrollobjekt der Klasse GEventControl jedes auftretende IE ab. Tritt nun ein Wechsel in den „normalen" *PLAY*-Modus ein, dann folgt eine Wiederabspielphase aller gespeicherten Events. Nach dieser Phase visualisiert das System IEs wieder ohne Verzögerung. Wir gehen weiter unten (S. 86) im Detail auf diesen Vorgang ein.

5.3.2 Verarbeitung der Programmpunkteinstellungen

Entspricht der aktuell bearbeitete Programmpunkt einer Schleife oder einer GANILA-Annotation, dann kann die dispatch()-Methode der Klasse GAlgorithm dessen PPEs vor seiner Ausführung abrufen und in den Ablauf des Algorithmus (und damit auch in seine Animation) eingreifen. Das Laufzeitverhalten der meisten Annotationen (z. B. des Paralleloperators oder des Invariantenkonstrukts, ...) sowie die Animationssteuerung von Schleifen wurden bereits ausführlich in Abschnitt 3.2.4 über die Reifikation ausgewählter GANILA-Konstrukte diskutiert. Dabei gingen wir auch auf die Funktionsweise der in diesem Kontext wichtigsten Laufzeitmethoden ein, z. B. loopBegin() für die Schleifensteuerung oder executeParallel() für die parallele Ausführung von Programmcode. Der größte Teil dieser Methoden wurde in der Kontrollklasse GAlgorithm definiert. Erläuterungen tiefergehender Implementierungsaspekte würden in dieser Arbeit zu weit führen und das Veständnis des Lesers für die Zusammenhänge des Laufzeitsystems behindern. Allerdings gibt es noch offene und interessante Fragestellungen hinsichtlich unserer Implementierung des Konzepts der Interesting Events, die wir im folgenden kurz betrachten: das Zeitmanagement, die Aufzeichnung von IEs und die richtige Auswahl einer Teilmenge von IEs, die im Rahmen der Animationssteuerung von Schleifen zu visualisieren ist.

PPEs für Interesting Events

In Abschnitt 3.2.4 wurde anhand eines kleinen Codebeispiels die Übersetzung und Reifikation von IEs deutlich gemacht. Zur Wiederholung geben wir an dieser Stelle noch einmal den generierten JAVA-Code für ein IE *IE_Prime(n) mit Programmpunkt 4 an, welches innerhalb eines Methodenrumpfes durch den Visualisierer vorgegeben wurde:

```
public void dispatch_4(Gint a1) {
  GEvent e = new GEvent(4,
      isRecord(),
      isVisibleEvent(),
      getLoReDepth(),
      "Prime", new Object[]{ a1 });
```

```
    control.broadcast(e);
}
```

Die Methode dispatch_4() ist eine Klassenmethode des Algorithmenmoduls und erzeugt zur Laufzeit ein Eventobjekt der Klasse GEvent. Der Konstruktor dieser Klasse speichert eine Reihe von Informationen ab: den aktuellen Animationsmodus, den Aktivierungszustand des IE, die Umgebung für die Animationssteuerung von Schleifen, den Eventnamen und das Argument. Ein Kontrollobjekt control des Typs GEventControl sendet das Eventobjekt anschließend über einen Broadcast an alle registrierten Sichten. Diese und alle nachstehend beschriebenen Implementierungsaspekte der Verarbeitung von IEs sind im Klassendiagramm in Abbildung 5.5 veranschaulicht.

Kommunikation mit Sichten Jede Sicht erbt von einer Klasse SuperView und mit dieser eine Methode processEvent() für den Aufruf der in der Sicht zu implementierenden Eventhandler. Dieser Ansatz realisiert unmittelbar das Entwurfsmuster *Strategy* [GHJV95], dessen Vorteil es ist, mehrere Untersichten zu verwalten und beliebig miteinander austauschen zu können. Die broadcast()-Methode des Kontrollobjekts ruft diese Methode mit dem Eventobjekt als Argument für jede registrierte Sicht auf. Daraufhin erzeugt der im Methodenrumpf von processEvent() enthaltene Code für jede Sicht einen eigenen Thread. In diesem Thread wird über JAVA-Reflection ein geeigneter Eventhandler (s. u.) aufgerufen, der Animationen in der Sicht ausführt. Die Laufzeitumgebung wartet mit dem Aufruf des nächsten reifizierten Programmpunkts so lange, bis jede Sicht seinen Eventthread beendet hat. Dies gilt auch für Sichten, deren Eventthreads neue Unterthreads erzeugen und starten. Somit ist sichergestellt, daß jede Sicht ihre Animationen vollständig abgearbeitet hat, bevor mit dem nächsten Programmpunkt fortgefahren wird.

Die PPE für ein IE erlaubt zwar seine Deaktivierung zur Laufzeit der Algorithmenanimation, aber nicht während seiner eigenen Abarbeitung. Wenn ein Benutzer eine GANILA-Annotation ab- oder anschaltet, findet eine Veränderung der PPEs nur dann unmittelbar statt, wenn es sich nicht um den aktuellen Programmpunkt handelt. Andernfalls wird die Annotation noch mit den alten PPEs durchgeführt. Eine derartige Möglichkeit der (De-)Aktivierung von IEs führt in einer Sicht zur notwendigen Bereitstellung von zwei Eventhandlern pro Event: IE_*Name*_Play_Invisible() und IE_*Name*_Play_Visible(). Beide produzieren interne Zustandsänderungen der Sicht, aber nur der Eventhandler IE_*Name*_Play_Visible() erzeugt für das aktive IE eine graphische Ausgabe. Wie wir bereits in Abschnitt 3.2.4 auf S. 39 gesehen haben und auch weiter unten noch einmal aufgreifen werden, sind insgesamt sogar vier Eventhandler nötig, falls verschiedene Animationsmodi benutzt werden. Genau genommen haben damit die Einstellungen anderer Programmpunkte Einfluß auf die Verarbeitung von IEs. Die Sichten müssen aber nicht zwangsläufig auf alle in einem GANILA-Eingabeprogramm spezifizierten IEs reagieren. Falls eine Sicht keinen Eventhandler für ein spezielles IE bereitstellt, schlägt dessen Aufruf über processEvent() fehl, das System fängt die Ausnahme ab und läuft weiter. Für den Benutzer ist dieser Prozeß transparent.

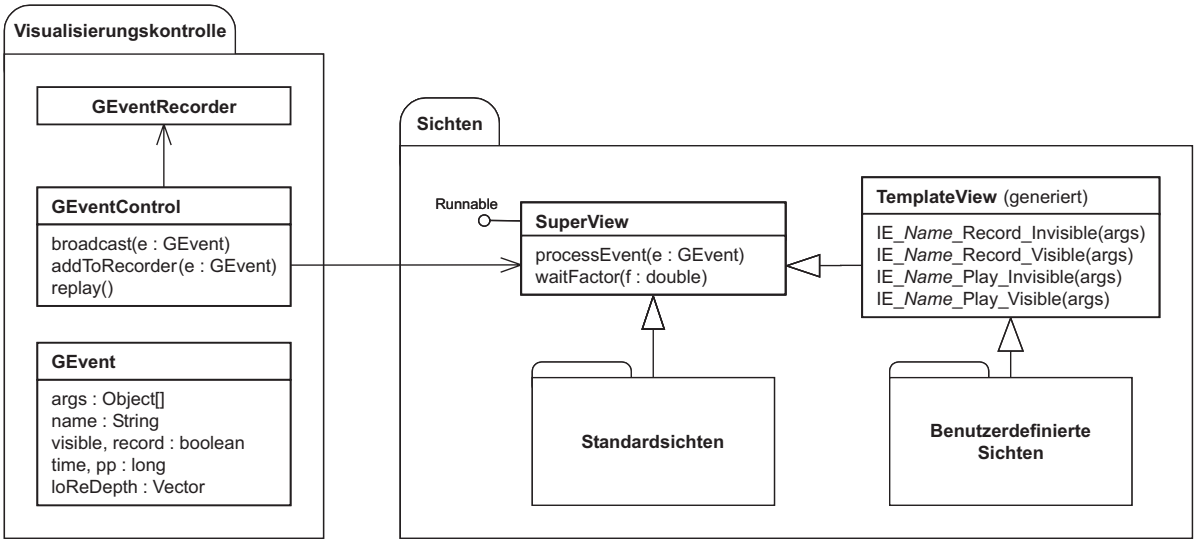


Abbildung 5.5: Kontrolle über die Verarbeitung von Interesting Events und deren Ausführung in den Sichten (UML-Notation).

Bisher wurden in diesem Prozeß der Eventbehandlung lediglich der Name des IE für die richtige Zuordnung zu den Eventhandlern, seine Argumente[4] und sein Aktivierungszustand ausgenutzt. Das Eventobjekt speichert jedoch noch weitere Informationen, wie im Codebeispiel zu Beginn des Abschnitts bereits zu sehen war. Im folgenden beleuchten wir deren Verwendung für weitere Aspekte der Visualisierungskontrolle von IEs.

Aufzeichnung Eine der im Eventobjekt gespeicherten Informationen ist der im Algorithmenmodul aktuell festgestellte Animationsmodus. Im *PLAY*-Modus findet der Verarbeitungsprozeß wie oben beschrieben statt. Im *RECORD*-Modus ruft die Methode processEvent() in Abhängigkeit vom derzeitigen Aktivierungszustand einen der beiden jetzt zusätzlich zu definierenden Eventhandler IE_*Name*_Record_Invisible() oder IE_*Name*_Record_Visible() auf. Für benutzerdefinierte Sichten (vgl. Abschnitt 5.4.1 u. Abb. 5.5) erzeugt der Compiler GAJA eine Templateklasse mit vordefinierten Methoden dieser Art, von der die Sichten erben müssen. Diese beiden Eventhandler produzieren keine visuelle Ausgabe und führen keine Änderungen am internen Zustand der Sicht durch, sondern berechnen die für eine post mortem-Visualisierung benötigten Daten. Sind diese Berechnungen beendet, speichert die Methode addToRecorder() das GEvent-Objekt in einem Rekorder der Klasse GEventRecorder ab (siehe Abb. 5.5). Dabei wird der Animationsmodus für dieses Eventobjekt auf *PLAY* zurückgesetzt, um keinen Zyklus in Gang zu setzen.

Erreicht die Animation einen Programmpunkt mit der Einstellung *PLAY*, so setzt eine Wiederabspielphase aller auf dem Rekorder gespeicherten Eventobjekte ein. Dazu entnimmt die Methode replay() des Kontrollobjekts control dem Rekorder nacheinander

[4] Argumente können auch Variablen mit gewrappten primitiven Datentypen sein. Diese werden geklont, um Seiteneffekte zwischen Algorithmenmodul und Sichten auszuschließen.

die einzelnen Eventobjekte und verschickt sie über den Broadcast-Mechanismus an alle Sichten. Ist der Rekorder vollständig geleert, läuft die Animation bis zum nächsten Moduswechsel im *PLAY*-Modus weiter.

Zeitmanagement und Synchronisation Das bisher beschriebene Zeitmanagement umfaßt nur eine sehr grobe Synchronisierung von IEs: Jedes IE läuft pro Sicht in einem eigenen Thread, und das Laufzeitsystem wartet so lange auf die Fortsetzung des Algorithmus, bis jeder Eventthread beendet worden ist. In der Gesamtkonzeption des GANIMAL-Frameworks ist keine globale Uhr vorgesehen, d. h. wir können nicht festlegen, daß beispielsweise eine Pfadanimation eines graphischen Objekts von Punkt A nach Punkt B auf jeder Plattform mit konstanter Framerate in exakt einer Zeitspanne t durchgeführt wird. Ein Grund für diesen Verzicht liegt u. a. darin, daß es bei live/online-Animationen in unserem Framework nicht möglich ist, die Laufzeit einer Operation vorherzusagen.

Dies ist ein besonderer Nachteil, wenn der Visualisierer mehrere Sichten für ein IE implementieren will. Komplexere Animationen, die durch die verschiedenen Eventhandler jeder Sicht ausgeführt würden, wären voneinander losgelöst. Obwohl die Sichten den gleichen Sachverhalt graphisch vermitteln sollen, sind evtl. einige in ihrer Animation weiter fortgeschritten als andere oder haben schlimmstenfalls ihren Eventthread bereits beendet, während die Animation in anderen Sichten noch andauert.

Unser Ansatz, mit dem wir dieser Problematik entgegentreten, basiert auf der Idee, jedem IE eine bestimmte, feste Zeitspanne $\mathsf{t_{IE}}$ zur Verfügung zu stellen. Die entsprechenden Eventhandler der verschiedenen Sichten haben, sofern es die Rechenleistung zuläßt, genau diese Zeit zur Verfügung, um ihre Operationen auszuführen. Ihr Programmcode wird an geeigneten Stellen mit einem Methodenaufruf waitFactor(f) der Klasse SuperView annotiert. Der Faktor f mit $0 < f \leq 1$ besagt, daß zum Zeitpunkt dieses Methodenaufrufs die Zeit $f \cdot \mathsf{t_{IE}}$ zwingend verbraucht wird. Annotieren die Eventhandler zweier Sichten z. B. ihren Code mit waitFactor(0.5), dann verbrauchen beide Sichten ihre (identische) Eventzeit $\mathsf{t_{IE}}$ bis zur Hälfte — ggf. muß eine Sicht etwas warten — und setzen ihre weitere Animation „gleichzeitig“ fort. Versuche haben gezeigt, daß der Scheduler der JAVA-API diesen Ansatz gut unterstützt.

Die Realisierung dieses Ansatzes wird im folgenden beschrieben. Ein Eventobjekt speichert im Moment seines Broadcasts die aktuelle Systemzeit $\mathsf{t_{start}}$. Anschließend rufen die Eventhandler der Sichten die Methode waitFactor() mit entsprechenden Argumenten auf. Betrachten wir die Implementierung dieser Methode genauer:

```
public synchronized long waitFactor(double f) {
  long t'IE = max(1, tIE * speed);
  long t = f * t'IE − (tcurrent − tstart);
  if(t > 0) { Thread.currentThread().sleep(t); }
  return t;
}
```

Die erste Zeile im Rumpf der Methode berechnet über einen Aufruf der JAVA-API-Methode max() eine Zeit $\mathsf{t'_{IE}}$, die von der aktuellen Geschwindigkeitseinstellung des in

der GUI vorhandenen Schiebereglers abhängt. Nach der Multiplikation mit dem o. g. Faktor f wird von der errechneten Vorgabezeit die tatsächlich verbrauchte Zeit subtrahiert. Ist das Ergebnis kleiner oder gleich Null, dann hat der Eventthread zuviel Zeit verbraucht und die Methode liefert das Ergebnis zurück. Der Programmierer einer Sicht könnte aufgrund des Rückgabewerts seinen Programmcode derart gestalten, daß z. B. Zwischenschritte einer Animation in einem solchen Fall unterdrückt werden, um Zeit wieder aufzuholen. Ist das Ergebnis größer Null, dann wird der Eventthread einer Sicht für diese Zeit pausiert. Für die meisten Animationen reicht dieser Ansatz aus, und er paßt sich der Rechnerleistung an: Auf sehr schnellen Rechnern laufen Algorithmenanimationen bei starker Zeitsegmentierung durch die Methode waitFactor() und bei gleicher Geschwindigkeitseinstellung nicht zu schnell ab.

Auswahl der zu animierenden Schleifeniterationen Der Abschnitt über die Übersetzung von Schleifen mit Visualisierungsbedingung in Kapitel 3 (S. 39 ff.) beschrieb die Eigenschaften einer Reihe von Laufzeitmethoden. Deren Programmlogik greift tief in den Verarbeitungsprozeß von IEs ein. Der Vollständigkeit halber wiederholen wir hier kurz die wichtigsten Aspekte. Als Resultat der diskutierten Vorgehensweise wurde eine Umgebung mit den endgültigen Werten für die Variablen $i und $n in allen aufgezeichneten Eventobjekten in Form eines Vektors abgespeichert (siehe Abb. 5.5). Diese Umgebung beinhaltet neben den o. g. Variablen noch die Programmpunkte der Schleifen, die eine Visualisierungsbedingung aufweisen und in deren Schleifenrümpfen sich die ursprünglichen IEs befinden. Mit diesen Informationen kann eine weitere (überladene) Methode replay() des Kontrollobjekts alle Visualisierungsbedingungen in der Wiederabspielphase der IEs abtesten. Dazu wird für jeden Umgebungseintrag (pp, i, n) die von GAJA generierte Testmethode st[*pp*].checkVisits(i, n) ausgeführt. Diese ist Bestandteil der PPEs der Schleife *pp*. Liefert sie den Booleschen Wert **true** zurück *und* ist das Eventobjekt eines Events ***IE**_*Name*() als aktiv gekennzeichnet, dann wird dieses über den Broadcast-Mechanismus letztendlich auch von IE_*Name*_Play_Visible() ausgeführt; in allen anderen Fällen von IE_*Name*_Play_Invisible().

5.4 Sichten

Die GANIMAL-Laufzeitumgebung setzt Sichten für die Visualisierung von Interesting Events ein. In unserem System unterscheiden wir zwei Arten von Sichten:

- *Benutzerdefinierte Sichten* werden vom Visualisierer für eine bestimmte Algorithmenanimation speziell entwickelt.
- *Standardsichten* sind fester Bestandteil der Laufzeitumgebung und können vom Visualisierer in eine Animation eingebunden werden.

Alle in einer Animation verwendeten Sichten sind im GANILA-Programmcode zu initialisieren. Dies geschieht im Fall der Standardsichten und der von ihnen erbenden

Sichten über die Angabe einer Sichtendeklaration vor der Klassendefinition des Algorithmus und nach den in Java-Programmen üblichen Importdeklarationen (siehe dazu auch Abschnitt 3.1.1, in dem die verschiedenen Ganila-Konstrukte diskutiert wurden). Benutzerdefinierte Sichten, die nicht von einer Standardsicht erben, können über eine separate Klasse initialisiert werden, wie wir weiter unten erläutern werden.

5.4.1 Benutzerdefinierte Sichten

Die Möglichkeit der Implementierung von benutzerdefinierten Sichten gibt dem Entwickler einer Animation die Freiheit, selbst die Entscheidung darüber zu treffen, auf welche IEs eine Sicht reagieren und wie diese visualisiert werden sollen. Da auch diese Art von Sichten für jedes IE vier verschiedene Eventhandler definieren muß, generiert der Compiler GaJa zur Unterstützung des Entwicklers eine Templateklasse. Diese enthält leere Methodendefinitionen der Eventhandler. Mittels Vererbung können die benutzerdefinierten Sichten die Methoden bei Bedarf überschreiben, etwa wenn die Erzeugung einer erweiterten post mortem-Visualisierung beabsichtigt wird (vgl. S. 39). Abbildung 5.6 zeigt die Vererbungshierarchie der verschiedenen Sichten.

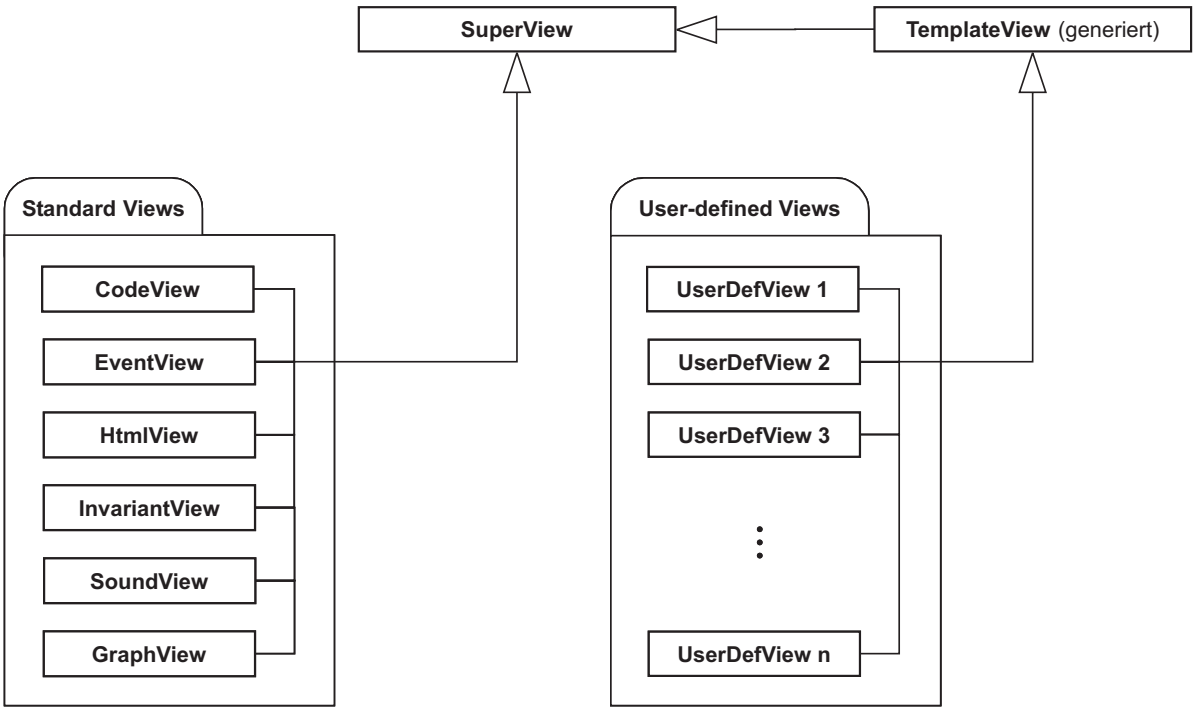


Abbildung 5.6: Sichten im Ganimal-Laufzeitsystem (UML-Notation).

Im Normalfall entwickelt der Visualisierer eine neue Sicht für einen speziellen Algorithmus und läßt diese, wie oben beschrieben, von einer algorithmenspezifischen Templatesicht erben. Die Laufzeitumgebung enthält ein Basispaket mit graphischen Grundfunktionen. Dieses besteht aus einer Menge von Java-Klassen, die primitive Methoden zur Kommunikation, zum Zeichnen und für Animationen beinhalten, wie z. B. Fenster,

graphische Grundformen, Animationen entlang eines festgelegten Pfades etc. Sämtliche benutzerdefinierten Sichten sollten dieses Paket in ihrer Implementierung verwenden, da auf diese Weise ein konsistentes Erscheinungsbild der verschiedenen Sichten gefördert wird.

Um benutzerdefinierte Sichten korrekt in der Laufzeitumgebung anzumelden, ist eine zusätzliche Klasse zu implementieren, in der die Visualisierungskontrolle eine Registrierung dieser Sichten erhält. Weiterhin können dort Layouteigenschaften der Zusammenstellung mehrerer Sichten definiert werden. Im allgemeinen lassen sich alle Sichten beliebig ineinander verschachteln und sind selbst für das korrekte Eventhandling innerer Sichten verantwortlich. Diese Eigenschaft des Laufzeitsystems wird durch das in Abschnitt 5.1 erwähnte Entwurfsmuster *Composite* realisiert. Im nächsten Kapitel gehen wir in Abschnitt 6.2.1 auf die Animation des Heapsort-Algorithmus ein und geben einen Überblick über die Implementierung einer benutzerdefinierten Sicht. Der Visualisierer kann eine neue Sicht auch derart gestalten, daß diese von einer Standardsicht erbt. Dieser Fall wird im folgenden Abschnitt näher beleuchtet.

5.4.2 Standardsichten

Der Entwickler von Animationen kann gegenwärtig unter sechs vordefinierten Standardsichten wählen und sie unverändert übernehmen. Diese unterstützen bereits viele Visualisierungen von Programmzuständen und Datenstrukturen und weisen Eigenschaften auf, die in Algorithmenanimationen häufig benötigt werden. So lassen sich recht schnell komplexe Animationen einfacher Algorithmen erstellen.

Reichen die vorhandenen Funktionalitäten nicht aus, dann besteht neben der Implementierung benutzerdefinierter Sichten auch die Möglichkeit, Standardsichten durch Vererbung zu erweitern bzw. zu verändern. Hierbei ist zu beachten, daß in unserem Framework ein Eventobjekt für ein IE auf zwei unterschiedliche Arten erzeugt werden kann: Die erste Art umfaßt Events, die in der Sprache Ganila im Eingabeprogramm spezifiziert werden, d. h. die vom Visualisierer eingefügt und zur Laufzeit des Algorithmus an *alle* Sichten verschickt werden. Es ist leicht möglich, eine „normale“ benutzerdefinierte Sicht so zu erstellen, daß sie die gleichen Eventhandler wie eine bestimmte Standardsicht implementiert. Eine Teilmenge der Standardsichten verarbeitet aber spezielle Events, die nicht durch den Visualisierer, sondern durch das System initiiert werden. So schickt die Visualisierungskontrolle beispielsweise nach jedem Programmpunkt eine Nachricht in Form eines Events an eine sog. CodeView (s. u.), die mit dieser Information den aktuellen Programmpunkt in einem separaten Fenster markiert. Diese zweite Art von IE wird nicht an alle registrierten benutzerdefinierte Sichten weitergeleitet, sondern nur an die jeweilige Standardsicht. Über Vererbung erhalten neue Sichten den gleichen (allgemeineren) Klassentyp, und die Laufzeitumgebung kann auf diesem Weg systeminitiierte Events an eine neue Sicht senden.

CodeView

Die CodeView ermöglicht eine textuelle Sicht auf das auszuführende Programm und zeigt den aktuell ausgeführten Programmpunkt mit Hilfe einer Farbmarkierung an. Im Unterschied zur GUI werden hier die Definitionen der IEs sowie alle sonstigen Sprachkonstrukte von GANILA ausgeblendet. Ist der aktuelle Programmpunkt eine GANILA-Annotation, bleibt der vorhergehende Programmpunkt farblich unterlegt. Abbildung 5.7 zeigt die CodeView, während sie den Programmcode eines Algorithmus anzeigt. Sie implementiert einen Eventhandler für das folgende Event[5], welches nur von der Visualisierungskontrolle verschickt wird:

- *ShowPP(Programmpunkt)*. Es bewirkt die Markierung des aktuellen Programmpunkts in einer Signalfarbe.

Das nächste IE kann durch den Visualisierer im GANILA-Eingabeprogramm, d. h. über das Konstrukt ***IE_Flash()**, gesetzt werden:

- *Flash()*. Der Hintergrund der CodeView blinkt mehrmals kurz auf. Dieses Verhalten ist besonders dann von Nutzen, wenn auf interessante Stellen im Programmcode aufmerksam gemacht werden soll.

EventView

Wird zur Laufzeit des Algorithmus ein in GANILA definiertes IE erreicht, so listet die EventView dieses Event einschließlich seiner Argumenttypen und -werte in einem separaten Fenster auf. Man erhält so eine History aller aufgetretenen Events, wie in Abbildung 5.8 gezeigt wird. Systeminitiierte Events werden ignoriert. Die EventView entspricht der sog. ScriptView des ZEUS-Algorithmenanimationssystems [Bro91] und unterstützt den Visualisierer zusätzlich bei der Fehlersuche. Es existieren ausschließlich Eventhandler für systeminitiierte IEs, d. h. der Visualisierer muß bei Bedarf nur die Deklaration der EventView im Kopf der GANILA-Datei angeben.

- *AddEvent(Name, Argumente)*. Das System sendet den Namen und die Argumente (Typ und Wert) an die EventView, in der diese Informationen textuell dargestellt werden.

- *AddEventLine(Name, Argumente)*. Dieses Event hat die gleiche Funktionalität wie *AddEvent*, fügt aber noch einen Zeilenumbruch hinzu.

[5] Abstrakte Beschreibungen von IEs werden *kursiv* abgedruckt. Die aufgezählten Beschreibungen machen hier die Ausdrucksstärke der verschiedenen Standardsichten deutlich.

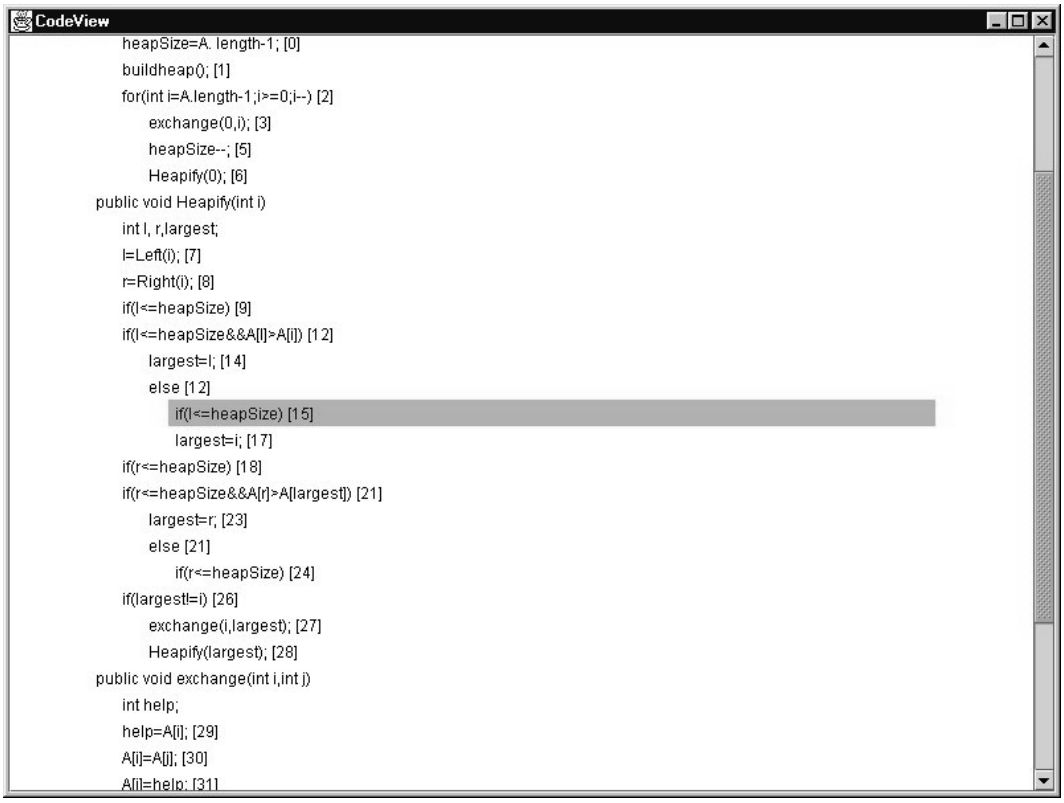


Abbildung 5.7: Bildschirmaufnahme der CodeView.

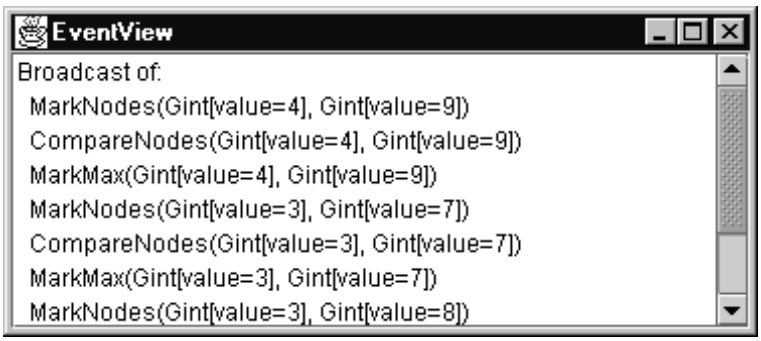


Abbildung 5.8: Bildschirmaufnahme der EventView.

HtmlView

Über IEs für die HtmlView lassen sich Programmpunkte mit HTML-Dokumenten verbinden. Die Sicht zeigt also eine beliebige HTML-Seite über die Angabe einer URL (Uniform Resource Locators) als Argument eines Events an. Dazu wurde ein JAVA-basierter Webbrowser *IceBrowser 4.08* [Ice02] in das Laufzeitsystem eingebettet.

Diese Eigenschaft erlaubt eine Reihe von nützlichen Anwendungen in der Dokumentation des zu animierenden Algorithmus: In GANILA kann die Dokumentation eines Programmpunkts angezeigt werden, wann immer dieser während der Animation erreicht wird. Darüber hinaus können Dokumentationen selbst erstellt und mit der Animation gleichsam mitgeliefert, oder bereits vorhandene, im WWW zu findende Dokumentationen verwendet werden. Betrachten wir das in Abbildung 5.9 präsentierte Beispielszenario für die Animation des Heapsort-Algorithmus: Die HtmlView zeigt grundlegende Definitionen zu diesem Sortierverfahren an, wie etwa zur Heapeigenschaft. Die Animation des Algorithmus könnte diese Seite vor der Phase der ersten Heapbildung aufrufen. Im Verlauf der Animation kann über die Angabe von Ankern auch zu bestimmten Teilbereichen der Seite gescrollt werden, wie Abbildung 5.10 verdeutlicht. Platziert der Entwickler ein entsprechendes IE vor den Aufruf der Methode buildheap() (vgl. Anhang A), so kann das Laufzeitsystem die Animation anhalten und die Webseite in der HtmlView anzeigen. Der Benutzer hat in unserem Beispiel sogar die Möglichkeit, mit Hilfe eines in die Seite eingebetteten JAVA-Applets selbst auszuprobieren, wie die Methode aus einem binären Baum einen Heap berechnet. Hierzu wählt er per Mausklick die Knoten mit maximaler Markierung aus, und das Applet gibt je nach Korrektheit seiner Aktion eine Fehlermeldung aus oder baut den Baum entsprechend um. So ist auch die Nutzung bereits vorhandener Ressourcen und anderer Technologien, etwa von Diagrammen, animierten GIFs oder von FLASH-Animationen [Mac02b] in einer GANIMAL-Animation möglich.

Die HtmlView gestattet über die Angabe von Parametern in ihrer Deklaration, ein neues Fenster zu öffnen, oder ein bereits geöffnetes Fenster zum Anzeigen des Dokuments zu verwenden. Weiterhin wird die Fenstergröße über Parameter der in GANILA zu spezifizierenden IEs bestimmt:

- *Open(ID, URL, Breite, Höhe)*. Dieses IE zeigt das Dokument zu der angegebenen URL in einem Fenster der angegebenen Breite und Höhe an. Durch die ID kann der Animator festlegen, ob das Dokument in einem bereits geöffneten Fenster oder in einem neuen Fenster angezeigt werden soll.

- *Close(ID)*. Hier wird das über das Event *Open* geöffnete Fenster mit der angegebenen Kennzeichnung wieder geschlossen.

- *Show(ID, URL, Breite, Höhe)*. Das Event stellt analog zu *Open* ein Dokument in einem Fenster dar. Dieses Fenster wird jedoch solange angezeigt, bis es durch die Betätigung einer Schaltfläche an seinem unteren Rand geschlossen wird.

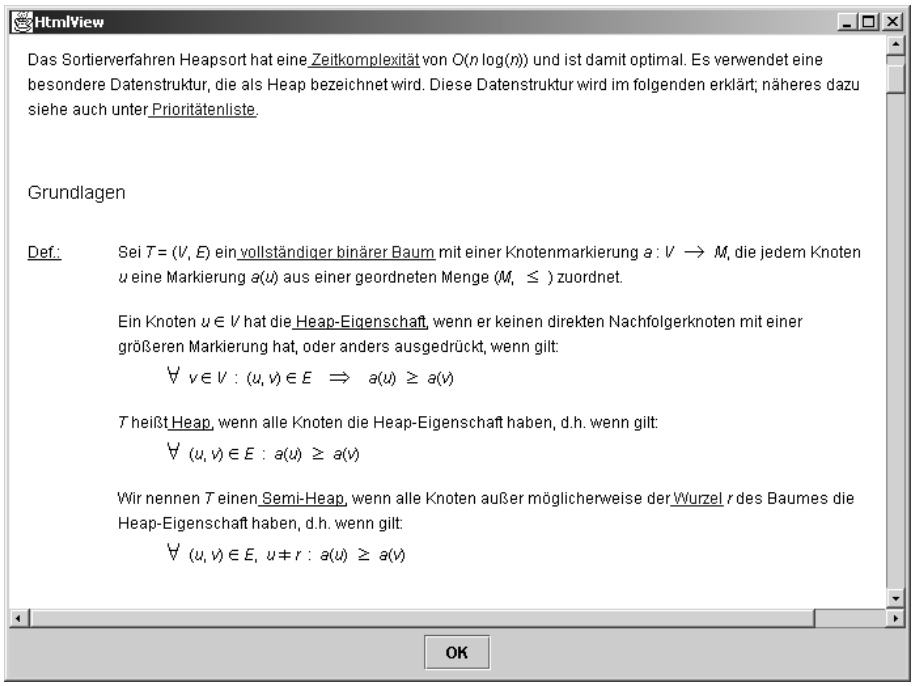


Abbildung 5.9: Bildschirmaufnahme der HtmlView, die eine externe Webseite [Lan02] mit einigen Definitionen zum Sortierverfahren Heapsort darstellt.

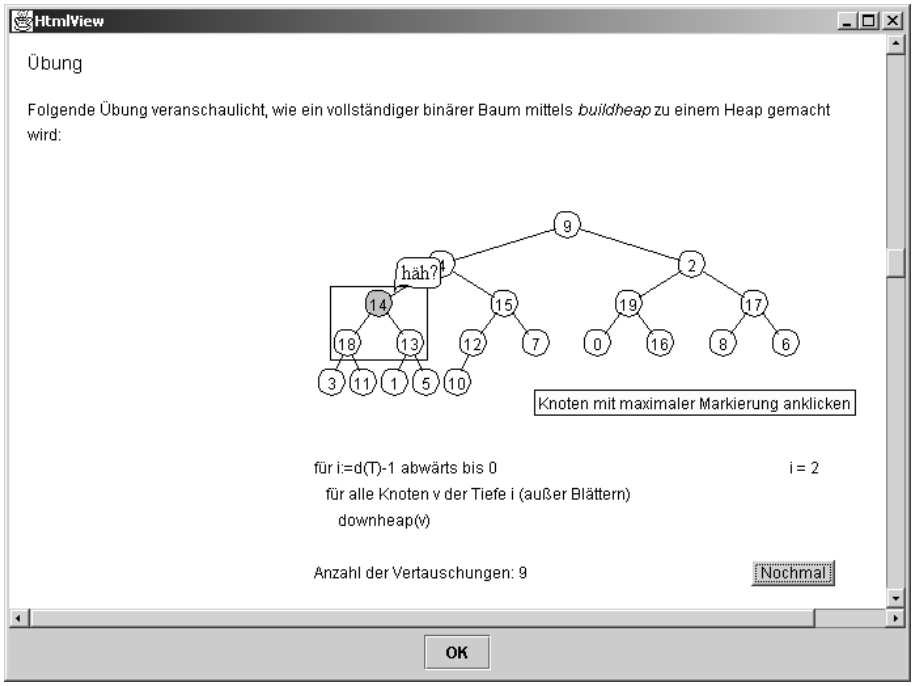


Abbildung 5.10: In der HtmlView läuft ein interaktives Java-Applet [Lan02] ab, mit dem Lernende die Bildung eines Heaps über Mausklicks üben können.

SoundView (Aura)

Diese „Sicht“ verbindet in einer zur HtmlView analogen Weise Programmpunkte mit Audiodateien. Audio kann für akustisches Feedback und gesprochene Erklärungen herangezogen werden. So weisen bestimmte akustische Signale auf wichtige Ereignisse hin und helfen, die Aktivitäten des Benutzers sinnvoll zu begleiten und ggf. sogar zu führen.

Die URLs aller Audiodateien sind in der Sichtendeklaration vorab anzugeben. Damit werden sie vor der eigentlichen Animation vom System geladen (siehe S. 25). Sonst kann es zu unvorhersehbar langen Ladezeiten kommen, wenn die Dateien auf einem entfernten Server liegen. Nimmt der Entwickler in einem GANILA-Programm eine Annotation zum Abspielen mehrerer Audiodateien vor, dann braucht er diese lediglich mit ihrem Dateinamen zu referenzieren, die eindeutig sein müssen. Die Lautstärke wird über die Optionen des installierten Soundtreibers eingestellt. Die SoundView reagiert auf die im folgenden beschriebenen IEs, wobei *ID* dem Dateinamen der betreffenden Audiodatei entspricht:

- *PlaySound(ID)*. Dieses IE spielt die Audiodatei einmal ab. Unterstützte Formate sind unkomprimierte lineare PCM-Audiodateien (Pulse Code Modulation-Audiodateien), wie z. B. WAV, AU, AIFF, MIDI oder RMF.

- *LoopSound(ID)*. Das Event spielt die Audiodatei bis zum Auftreten des IE *StopSound* mehrmals hintereinander ab.

- *StopSound(ID)*. Das Event unterbricht das Abspielen einer Audiodatei, welches ursprünglich durch die IEs *PlaySound* bzw. *LoopSound* gestartet wurde.

InvariantView

Gibt der Visualisierer über das GANILA-Konstrukt ***IV**(<*CondExpr*>) eine zu überprüfende Hypothese für einen Codeblock vor, dann bedarf es einer graphischen (evtl. auch auditiven) Darstellung ihrer Gültigkeit. Das GANIMAL-Laufzeitsystem realisiert diesen Anspruch mit einer InvariantView, die in Abbildung 5.11 auf Seite 97 dargestellt ist.

Sie besteht aus einer Tabelle, in der ineinandergeschachtelte Hypothesen von oben nach unten kellerartig eingetragen sind. Jede Hypothesenzeile ist in drei Bereiche untergliedert, welche eine eindeutige Kennzeichnung der Hypothese, die Hypothese *CondExpr* selbst sowie den Wahrheitswert für den aktuellen Programmpunkt enthalten. Der Wahrheitswert wird in jedem Animationsschritt durch das Laufzeitsystem überprüft und fortlaufend aktualisiert. Auf eine weitere Repräsentation des aktuellen Programmpunkts wurde in unserer Implementierung verzichtet, da dieser in der GUI farblich markiert erscheint. Bis auf die Annotationen hat der Visualisierer nichts zu tun. Falls er keine Deklaration der Sicht vornimmt, erzeugt der Compiler GAJA automatisch den Konstruktor für diese Sicht, sobald bei der Analysephase der Übersetzung ein ***IV**-Konstrukt entdeckt wird. Die InvariantView implementiert ausschließlich Eventhandler, die von der Visualisierungskontrolle gesendete IEs bearbeiten:

- *Start(PP des Konstrukts, Hypothese).* Dieses Event wird beim Betreten des von einem ***IV**-Konstrukt umschlossenen Blocks gesendet. Es fügt die angegebene Hypothese in die kellerartige Darstellung der InvariantView hinzu.

- *CheckPP(Liste von Wahrheitswerten).* Liegt der aktuell bearbeitete Programmpunkt im Gültigkeitsbereich mindestens eines Invariantenkonstrukts, dann überprüft das Laufzeitsystem die Erfüllbarkeiten der Hypothesen an diesem Programmpunkt. Anschließend sendet es der Sicht eine Liste mit entsprechenden Wahrheitswerten.

- *Close().* Wird der Block des innersten Invariantenkonstrukts wieder verlassen, so muß die entsprechende Hypothesenzeile wieder aus der Sicht entfernt werden.

GraphView

Zum Abschluß betrachten wir die wohl am häufigsten benötigte Sicht auf einen Algorithmus: die Visualisierung von Datenstrukturen. Die GraphView stellt dem Entwickler eine Auswahl von drei Graphlayoutalgorithmen zur Verfügung. Darunter fallen Baumlayouter, wie etwa ein einfaches Layoutverfahren, das ähnliche Ergebnisse liefert, wie die Dateidarstellung des Microsoft Windows Explorer (daher bekannt als Explorerlayout) bzw. wie das Layoutverfahren der Klasse JTree des JDK. Weiterhin wurde der Algorithmus von Walker [Wal90] für das Layout allgemeiner Bäume integriert. Graphen können mit Hilfe des Sugiyama-Verfahrens [STT81, San96] gezeichnet werden (vgl. hierzu auch Abschnitt 5.4.3.1).

Über die Parameter der Sichtendeklaration werden neben diesen Layoutern noch die Größe der GraphView, der Fenstertitel und die Hintergrundfarbe angegeben. Zur Manipulation des dargestellten Graphen durch den Visualisierer sind die folgenden IEs implementiert:

- *AddNode(ID, Beschriftung).* Das Event fügt einen Knoten mit ID und Beschriftung hinzu. Hierbei können die Knoten (fast) beliebige JAVA-Swing-Komponenten sein. Somit ist es beispielsweise auch möglich, einen Graphen in einem Knoten eines anderen Graphen zu zeichnen.

- *RemoveNode(ID).* Der angegebene Knoten wird mit seinen eingehenden und ausgehenden Kanten entfernt.

- *AddEdge(ID, Quelle, Ziel, Beschriftung).* Das Event fügt eine Kante hinzu. Die Parameter bestimmen die ID, den Ausgangsknoten, den Zielknoten und die Beschriftung der Kante.

- *RemoveEdge(ID).* Die angegebene Kante wird aus der GraphView entfernt.

- *ActionList(Aktionenliste).* Dieses IE bietet die Möglichkeit, in einem Animationsschritt mehrere Aktionen vorzunehmen, die den Graphen verändern. Als Parameter wird eine Liste übergeben, die alle Aktionen am aktuellen Graphen beschreibt.

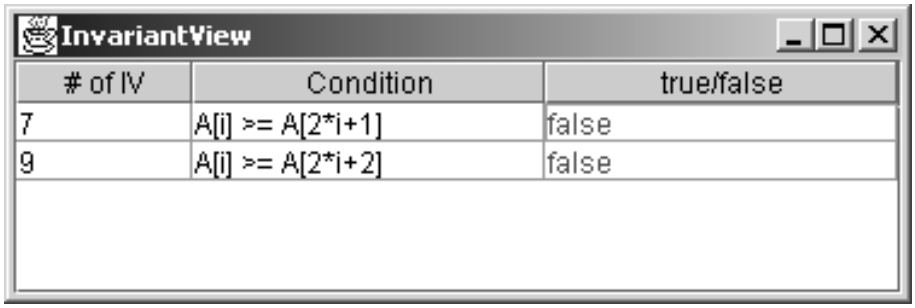


Abbildung 5.11: Bildschirmaufnahme der InvariantView.

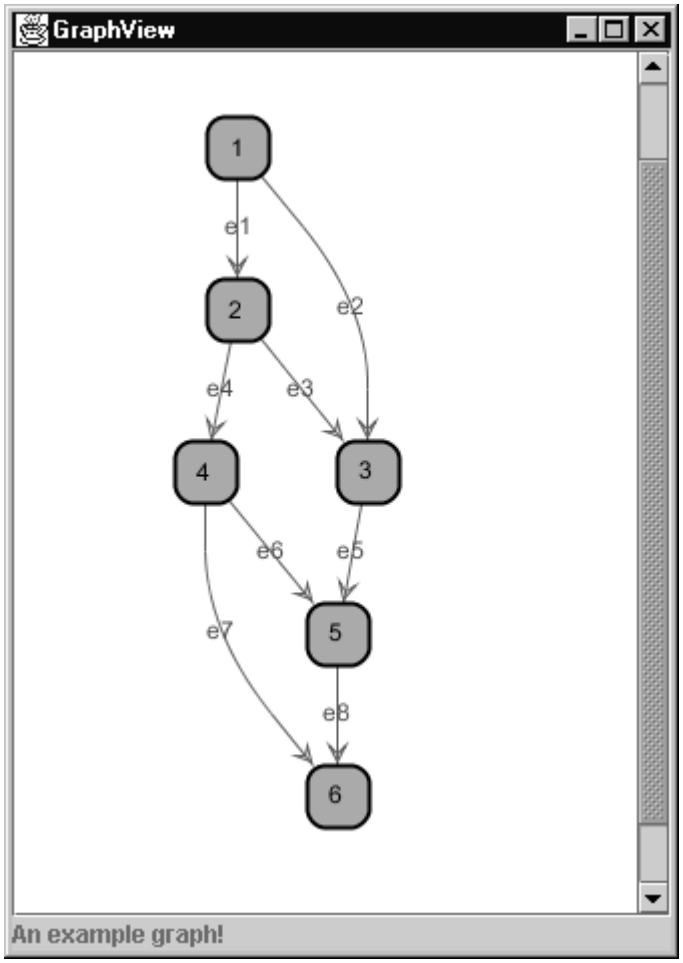


Abbildung 5.12: Bildschirmaufnahme der GraphView.

- *Clear()*. Das IE entfernt aus der GraphView alle graphischen Komponenten und stellt damit den Ausgangszustand wieder her.
- *Print(Text)*. Ein erläuternder Text wird in der Statuszeile des Fensters angezeigt.

Die GraphView unterstützt eine durch die Animationsmodi (S. 29 f.) kontrollierte post mortem-Visualisierung in besonderer Weise: Für IEs, die im *RECORD*-Modus aufgezeichnet worden sind, verwendet das System in der anschließenden Wiederabspielphase automatisch ein neues Layoutverfahren, bezeichnet als *Vorausschauendes Graphlayout*.

5.4.3 Vorausschauendes Layout von Graphen

Die meisten Arbeiten im Bereich des Graphenzeichnens richten sich auf das Layout eines einzelnen, statischen Graphen. Algorithmen wurden für verschiedene Klassen von Graphen (Bäume, DAGs, Digraphen, ...) und nach verschiedenen ästhetischen Kriterien entwickelt, wie z. B. Minimierung der Kantenkreuzungen sowie Knicke bzw. Krümmung der Kanten, möglichst kleiner Flächenverbrauch der Zeichnung oder Symmetriemaximierung [DETT94, HMM00]. Allerdings existieren sehr viele Anwendungsgebiete, in denen die Zusammenhänge zwischen Objekten und die Objekte selbst dynamisch sind, z. B.:

- Animationen von Graphalgorithmen bzw. Algorithmen, die auf verzeigerten Datenstrukturen arbeiten,
- dynamische Visualisierungen von Ressourcenallokation in Betriebssystemen und im Projektmanagement,
- Netzwerkverbindungen oder
- die sich permanent verändernde Hyperlinkstruktur des WWW.

Dynamisches Graphenzeichnen zielt auf das Problem des Layouts von Graphen, die sich im Laufe der Zeit durch das Hinzufügen und Löschen von Kanten bzw. Knoten entwickeln. Dies führt zu einem zusätzlichen ästhetischen Kriterium, welches als „Preserving the Mental Map“ [MELS95] bekannt ist. Unter *Mental Map* versteht man die Struktur eines Graphen, die der Betrachter wahrnimmt und verinnerlicht.

Ein naiver Ansatz zur Berechnung eines dynamischen Graphlayouts würde vermutlich nach jeder Veränderung ein neues Layout für den ganzen Graphen bestimmen. Dazu könnte man Algorithmen anwenden, die für ein statisches Graphlayout entwickelt wurden. Allerdings produziert dieser Ansatz in den meisten Fällen ein Layout, das dem Kriterium, die *Mental Map* zu bewahren, nicht genügt.

Eine Lösung stellt die aus dem Bereich der Keyframe-Animationen bekannte Technik *in-betweening* dar, bei der der Computer die Details einer Bewegung auffüllt, die zuvor vom Benutzer exakt festgelegt wurde. In unserem Kontext berechnet eine Implementierung dieser Technik fließende (engl. „smooth“) Animationen zwischen aufeinanderfolgenden Graphen, d. h. Knoten springen nicht auf ihre neue Position im Layout, sondern bewegen sich kontinuierlich dorthin. Dieser Ansatz liefert brauchbare Resultate, wenn

nur einige Knoten ihre Position ändern bzw. ganze Knotencluster ohne tiefgreifende Änderung ihres inneren Layouts bewegt werden. In den meisten Fällen sind derartige Animationen in ihrer Ästhetik beeindruckend. Wenn jedoch sehr viele Knoten ihre alten Positionen verlassen, dann zerstört dieser Vorgang meist die *Mental Map* des Betrachters.

Einen weiteren Ansatz verkörpern sog. inkrementelle Algorithmen. Sie versuchen, das Layout einfach soweit lokal anzupassen, wie Veränderungen Platz benötigen. Im schlimmsten Fall aber muß das Layout des kompletten Graphen neu berechnet werden, falls hinreichend große Änderungen stattfinden.

Das durch unsere Forschungsgruppe entwickelte Layoutverfahren unterscheidet sich grundlegend von den bisher beschriebenen Lösungen. Wir betrachten allerdings nur Graphen, deren Änderungen bereits über einen bestimmten Zeitraum hinweg vor der Durchführung des Layoutverfahrens bekannt sind. Aus einer Sequenz von n Graphen berechnen wir ein globales Layout, welches für jeden der n Graphen ein eigenes Layout bewirkt. Die besondere Eigenschaft dieses Ansatzes ist, daß innerhalb einmal gezeichneter Graphen weder die Positionen der Knoten noch die Krümmung der vorhandenen Kanten in aufeinanderfolgenden Graphen verändert werden. Auf diese Weise wird die *Mental Map* des Betrachters beibehalten. Für die Berechnung des globalen Layouts ist ein (fast) beliebiger statischer Graphlayoutalgorithmus einsetzbar. Wir haben den Algorithmus *Foresighted Layout* genannt, da er die zukünftige Gestalt der Graphen kennt.

Im weiteren Verlauf dieses Abschnitts werden wir die Funktionsweise des Layoutverfahrens diskutieren, um uns im Anschluß zwei Anwendungsfällen zuzuwenden. Eine detaillierte Beschreibung des Ansatzes einschließlich seiner Implementierung wird in [DGK00, Gör01, DGK01] gegeben.

5.4.3.1 Beschreibung des Algorithmus

Als Eingabe für unser Layoutverfahren dient eine *Graphanimation* G, die als Folge von n Graphen $[g_1, \ldots, g_n]$ definiert ist. Anschließend wird aus der Vereinigung aller Knoten- und Kantenmengen aus G ein neuer Graph gebildet, den wir als *Supergraphen* g_G bezeichnen. Wird nun g_G durch einen statischen Layoutalgorithmus gezeichnet, können die so gewonnenen Layoutinformationen in der anschließenden Animation der Graphen ausgenutzt werden. Als Ergebnis erhält man ein dynamisches Layout, das die *Mental Map* des Betrachters sehr gut aufrecht erhält. Allerdings ist das erzeugte Layout, vom Blickwinkel klassischer Kriterien des statischen Layouts aus betrachtet, häufig ästhetisch nicht gelungen. Probleme bereiten zum Beispiel viele Leerflächen im Layout der dynamischen Animation, da jedem Knoten und jeder Kante — unabhängig davon, ob sie zur gleichen Zeit existieren, in der Anfangsphase der Animation für immer verschwinden oder erst am Ende erscheinen — verschiedene Koordinaten zugewiesen werden.

Eine mögliche Lösung dieser Probleme liegt darin, Knoten mit verschiedenen Lebenszeiten zusammenzufassen. Der Supergraph wird so in einen kompakteren Graphen transformiert, den wir *Graphanimationspartition* (kurz GAP) nennen. In einer GAP fallen also Knoten mit disjunkten Lebenszeiten zu einem Superknoten zusammen. Berechnet man nun ein neues Layout für die GAP und nutzt die Layoutinformationen wie oben

beschrieben für das Layout der Animation, so stellt man folgendes fest: Die klassischen ästhetischen Kriterien eines statischen Layouts werden wesentlich besser erfüllt; jedoch zu Lasten der Aufrechterhaltung der *Mental Map*, weil sich einige Knoten eine Position teilen und vom Betrachter möglicherweise nicht mehr unterschieden werden. Ferner können Multikanten durch das Verfahren selbst entstehen, d. h. Kanten mit gemeinsamen Quell- und Zielknoten. Die Animation zeigt diese zwar niemals gleichzeitig während ihres Ablaufs an. Sie wurden aber so gezeichnet, daß sie sich nicht überlagern. Das resultierende Layout von Multikanten ist somit für einen Betrachter nicht nachzuvollziehen.

Treten diese Umstände in einer konkreten Situation nicht übermäßig in den Vordergrund, dann könnten wir unser Layoutverfahren auf dieser Stufe abbrechen und die Animation so belassen. Andernfalls müßten in der gleichen Art und Weise wie die Knoten von g_G auch alle Multikanten der GAP zu einer Kante vereinigt werden, falls sie disjunkte Lebenszeiten haben. Genauer betrachtet sind in diesem Vereinigungsprozeß nur Multikanten involviert, die infolge der Berechnung von g_G und der GAP neu entstanden sind. Multikanten in der Eingabe G bleiben korrekterweise Multikanten, weil sie gleiche Lebenszeiten aufweisen. Den so neu erzeugten Graphen nennen wir *Reduzierte Graphanimationspartition* oder kurz RGAP. Die Verwendung von RGAPs in einer Animation von Graphen führt zu ästhetisch schönen Resultaten. Allerdings weicht das Layout auch hier mehr und mehr von unserer Idealvorstellung über die Beibehaltung der *Mental Map* ab.

Vorausschauendes Layout wurde in Java implementiert und in das Ganimal-Framework integriert. Es kann aber auch als eigenständiges Forschungsgebiet angesehen werden. Tiefergehende Details, wie etwa Beweise zur $\mathcal{NP}$-Vollständigkeit der Minimierung von GAPs bzw. RGAPs oder die lokale Optimierung der Kantendarstellung in RGAPs, und eine Reihe von Anwendungsbeispielen sind in der Diplomarbeit [Gör01] sowie im Beitrag [DGK01] nachzulesen. Eine Weiterentwicklung des Layoutverfahrens, die bisher nicht in das Ganimal-Laufzeitsystem integriert wurde, ist in dem Artikel [DG02] beschrieben.

5.4.3.2 Anwendung in der Algorithmenanimation

Entsprechend dem Fokus dieser Arbeit haben wir Vorausschauendes Layout in der Algorithmenanimation angewendet: Das GaniFA-Applet visualisiert und animiert mehrere Generierungsalgorithmen aus der Automatentheorie, einschließlich der Generierung eines nichtdeterministischen endlichen Automaten (NEA) aus einem regulären Ausdruck (RA). GaniFA wurde in ein elektronisches Textbuch über die Theorie endlicher Automaten eingebettet (siehe Abschnitt 6.3 für eine ausführlichere Beschreibung dieser Anwendung).

Endliche Automaten sind hier als Übergangsdiagramme repräsentiert, die im Verlauf des Generierungsprozesses schrittweise ihre Gestalt verändern. Es entsteht eine Folge von Diagrammen, d. h. eine Graphanimation, welche sich gut als Eingabe für unser Layoutverfahren eignet. Abbildung 5.13 zeigt die Unterschiede zwischen statischem ad hoc-Layout und Vorausschauendem Layout am Beispiel einer Animation der Generierung eines NEA aus einem regulären Ausdruck $(a|b)^*$. Insgesamt sind drei verschiedene

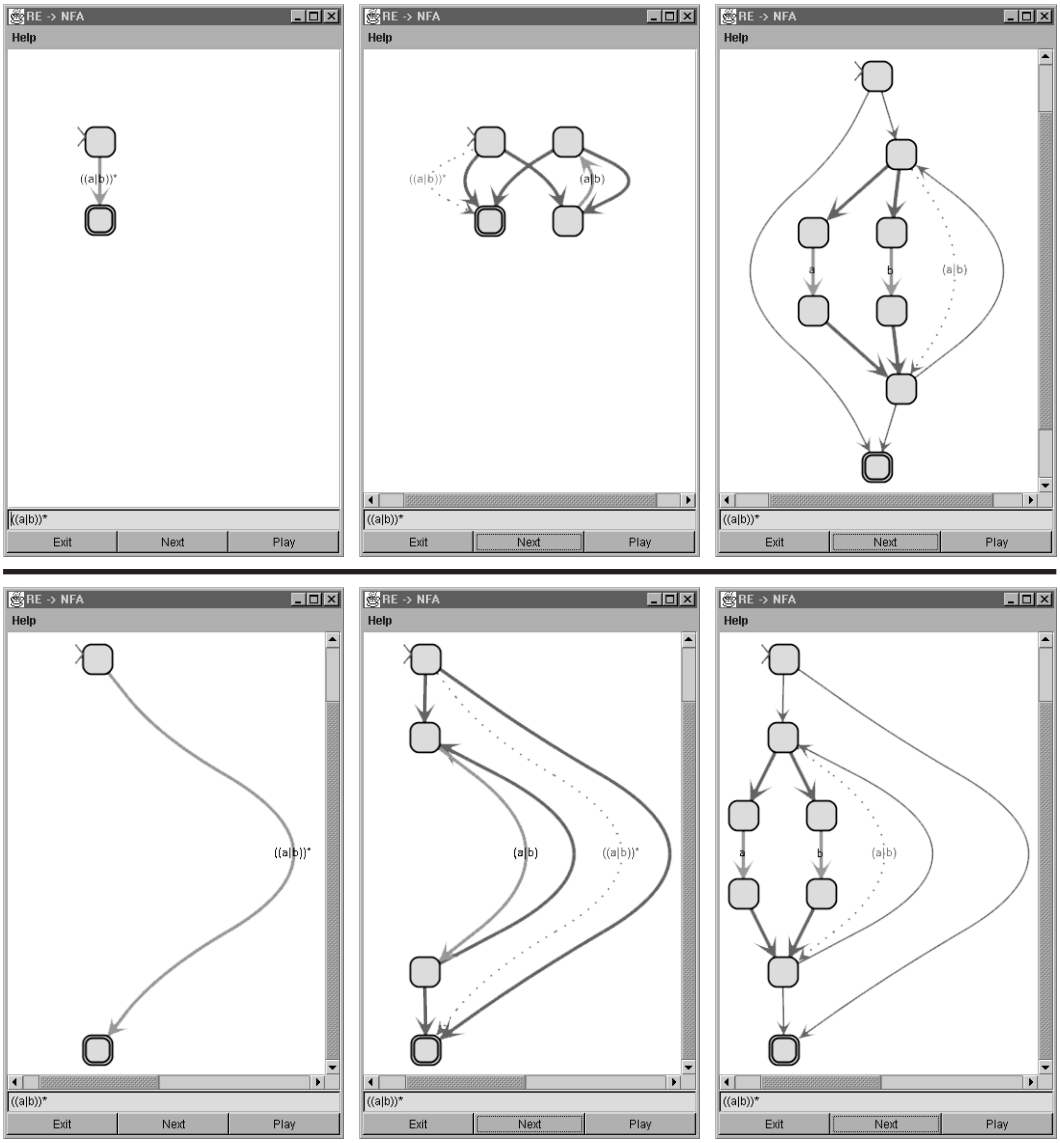


Abbildung 5.13: Ad hoc- und Vorausschauendes Layout einer Animation der Generierung eines NEA für den RA $(a|b)^*$.

Schritte sowie der endgültige NEA in chronologischer Abfolge von links nach recht dargestellt.

Das ad hoc-Layout im oberen Teil der Abbildung ist sehr verwirrend, da nach jeder Änderung ein neues Layout des aktuellen Graphen berechnet wurde. In jedem neuberechneten Layout wurden Knoten an verschiedene Positionen bewegt, obwohl der Algorithmus keinen dieser Knoten verändert hat. Das führt dazu, daß dem Benutzer nicht klar ist, welche Änderungen des Graphen durch das Generierungsverfahren selbst und welche Änderungen durch den Layoutalgorithmus herbeigeführt worden sind. Seine *Mental Map* über die Graphenstruktur ist „zerstört“. Der untere Teil der Abbildung zeigt, wie Vorausschauendes Layout dazu verwendet werden kann, diesen Nachteil zu vermeiden. Einmal kreierte Knoten und Kanten verlassen ihre angestammte Position nicht mehr, nachdem sie in den Graphen eingefügt worden sind. Diese Visualisierung des Generierungsverfahrens ist wesentlich klarer.

6 Anwendungen

In diesem Kapitel werden zwei Beispielanwendungen des GANIMAL-Frameworks vorgestellt. Bei der ersten handelt es sich um eine klassische Animation des Heapsort-Algorithmus, an der die grundsätzliche Vorgehensweise zur Erzeugung von Algorithmenanimationen mit der Sprache GANILA und dem GANIMAL-Laufzeitsystem deutlich wird. Das zweite Beispiel ist ein Applet, das die Generierung endlicher Automaten visualisiert sowie Berechnungen der generierten Automaten auf beliebige Eingaben hin animiert.

Vorher jedoch beleuchtet der nächste Abschnitt einige ältere Lernsysteme, die seit Mitte der 90er Jahre am Lehrstuhl für Programmiersprachen und Übersetzerbau an der Universität des Saarlandes entstanden sind. Die mit diesen Systemen gesammelten Erfahrungen sind in die Gestaltung und Implementierung des GANIMAL-Frameworks mit eingeflossen.

6.1 Vorausgegangene Entwicklungen

Allen präsentierten Lernsystemen gemein ist ihre inhaltliche Ausrichtung auf Problemstellungen aus dem Übersetzerbau. Neben der hier gegebenen Kurzdarstellung der abgedeckten Lerninhalte und der technischen Besonderheiten bzw. der Unterschiede zwischen den Systemen, kann eine lerntheoretische Klassifikation und eine didaktische Abgrenzung zum GANIMAL-Ansatz in Abschnitt 7.4 auf Seite 141 ff. nachgeschlagen werden.

6.1.1 Animation der lexikalischen Analyse

Die Lernsoftware „Animation der lexikalischen Analyse“ [BDKW99] (kurz ADLA) wurde mit dem Autorensystem Asymetrix Multimedia ToolBook 3.0 entwickelt und läuft unter Microsoft Windows ab der Version 3.1. Die Software bietet einerseits eine interaktive Einführung in die Probleme der lexikalischen Analyse, in welcher die wichtigsten Definitionen und Algorithmen in einer visuell ansprechenden Form präsentiert werden. Andererseits verdeutlichen statische Animationen, wie endliche Automaten aus regulären Ausdrücken generiert werden und wie diese arbeiten. Es wurde nur eine deutsche Version dieses Lernsystems entwickelt.

Der Benutzer navigiert sequentiell durch die einzelnen Lerneinheiten, die jeweils auf einer „Seite“ wie in einem Lehrbuch zusammengefaßt sind. Auf manchen Seiten besteht die Möglichkeit, Animationen zu starten und anschließend zu beobachten. Zunächst zeigen mehrere Animationen fundamentale Scannerkomponenten sowie die verschränkte

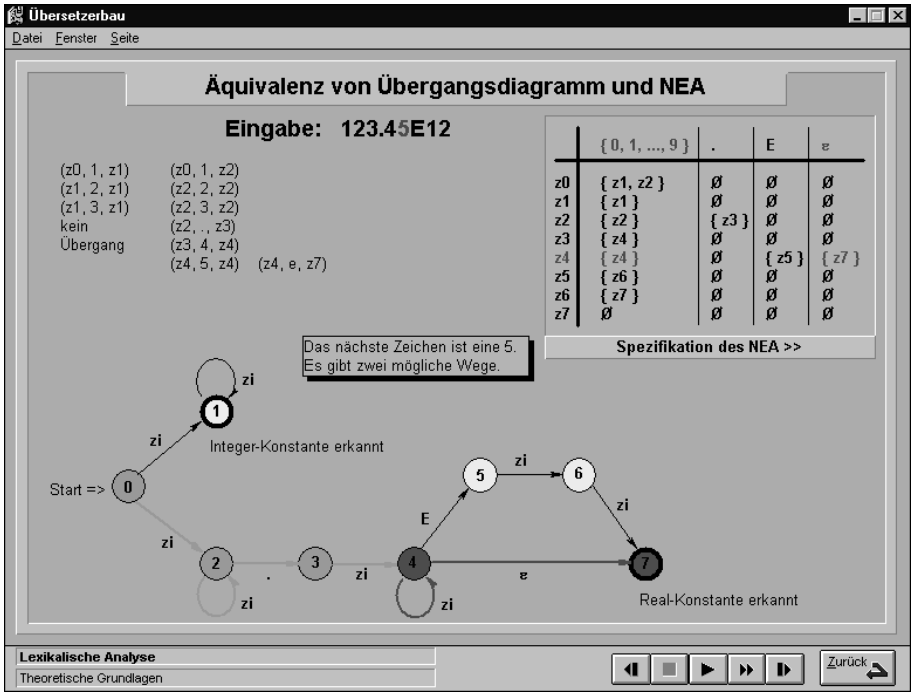


Abbildung 6.1: Äquivalenz von Übergangsdiagramm und NEA (AdlA).

Arbeitsweise zwischen Parser und Scanner. Dann werden die Begriffe, Symbole und Symbolklassen erläutert. Es wird darauf hingewiesen, wie Eingabesymbole, lexikalische Symbole, Symbolklassen und ihre interne Repräsentation zusammenhängen. Hierauf folgt eine Übersicht über formale Sprachen; insbesondere wird eine Einführung in die Theorie der regulären Sprachen und Ausdrücke gegeben.

Anschließend werden Übergangsdiagramme sowie nichtdeterministische (NEA) und deterministische (DEA) endliche Automaten beschrieben. Für all diese Konstrukte existieren animierte Beispiele, die durch den Benutzer steuerbar sind. Die Äquivalenz zwischen regulären Ausdrücken und NEAs wird anhand eines fest vorgegebenen, animierten Beispiels erklärt. Der Benutzer kann die parallele Abarbeitung eines Übergangsdiagramms und eines NEA auf demselben Eingabewort verfolgen. Wir sehen in Abbildung 6.1 die Bildschirmaufnahme eines NEA, der sich im Zustand $z4$ befindet. Das nächste zu lesende Zeichen ist das Zeichen 5. Nun kann der Automat das Zeichen 5 lesen, oder er kann einen Übergang via ε vornehmen. Die Animation stellt beide Möglichkeiten dar. Analog dazu wird der aktuelle Pfad im Übergangsdiagramm gekennzeichnet. Die zwei Kanten von Knoten 4 zu Knoten 7 sowie von Knoten 4 zurück zum selben Knoten 4 werden von dem System rot markiert. Die schattierte Box in der Fenstermitte beschreibt kurz, was der Automat bzw. das korrespondierende Übergangsdiagramm im aktuellen Schritt tut. In einem nächsten Schritt wird die Animation die mit dem Buchstaben E markierte Kante einfärben, die Erklärungsbox aktualisieren, den Zustand $z4$ als aktuellen Zustand markieren und den zweiten Übergang $(z4, \varepsilon, z7)$ verwerfen, da das nächste zu lesende Zeichen E ist.

Das AdlA-System vertieft weiterhin drei Algorithmen anhand von benutzerkontrollierbaren Animationen: die Transformation eines regulären Ausdrucks in einen NEA, eines NEA in einen DEA sowie eines DEA in einen minimalen DEA.

6.1.2 Animation der semantischen Analyse

Dieses System (Abk. AdsA) [Ker97, Ker99, Ker00] illustriert und animiert die Basisaufgaben der semantischen Analyse mit Hilfe textueller und graphischer Beispiele. Das vermittelte Basiswissen umfaßt die Konzepte von Gültigkeit und Sichtbarkeit, die Überprüfung der Kontextbedingungen (Identifikation von Bezeichnern, Überprüfung der Typkonsistenz), die Überladung von Operatoren und den Polymorphismus. Die korrespondierenden Analysealgorithmen können anhand eigener Beispiele untersucht werden. Wie das im vorhergehenden Abschnitt beschriebene System, wurde auch diese Anwendung mit Multimedia ToolBook 3.0 implementiert. Der dynamische Teil des Systems ist jedoch aufgrund der besseren Performanz und einiger Beschränkungen des ToolBook-Systems in C unter Verwendung der Windows-API implementiert. Eine solche Beschränkung ist z. B. die nicht akzeptable, limitierte Seitengröße für größere Bäume.

Zu Beginn präsentiert und beschreibt das System schrittweise die Definitionen der semantischen Analyse. Anschließend werden diese auf der Basis animierter Beispiele verdeutlicht. Beides geschieht vollständig interaktiv, d. h. die Benutzer können per Mausklick durch eine graphische Umgebung navigieren und Themen, für die sie sich interessieren, auswählen und vertiefen. Für diese Themen können sie erklärenden Text studieren und Animationen ansehen. Schließlich besteht die Möglichkeit, Beispiele selbst anzugeben, und die vorgestellten Algorithmen dynamisch auf den abstrakten Syntaxbäumen der eingegebenen Beispiele animiert ablaufen zu lassen. Beispiele können Eingabeprogramme, Ausdrücke oder Spezifikationen für die Überladung von Operatoren sein.

Statische Animationen Zunächst diskutieren wir den Teil des Lernsystems, in dem die Animationen und Visualisierungen auf fest vorgegebenen Beispielen beruhen, die vom Lernenden nicht modifiziert werden können. Dabei stellt das System die Aufgaben der semantischen Analyse vor, erläutert deren Problematik an vielen kleinen Animationen und bietet Problemlösungen anhand von Algorithmen an. Diese werden jeweils durch Algorithmenanimationen an einem festen, statischen Beispiel visualisiert.

In unserem ersten Beispiel (vgl. Abb. 6.2) soll die Funktionsweise eines Deklarationsanalysators zunächst an einer gedachten Übersetzungssituation beschrieben werden. Der mit einer Signalfarbe markierte Quelltext des Algorithmus kennzeichnet die aktuelle Stelle, deren Bedeutung auf der rechten Hälfte der Seite erklärt wird. Mit den Steuerungsknöpfen unten in der Kontrolleiste kann der Algorithmus komplett durchlaufen werden (Textanimation). Zu allen wichtigen Bestandteilen des Algorithmus lassen sich Beispielanimationen aufrufen, die ihrerseits wieder aus mehreren Seiten bestehen können.

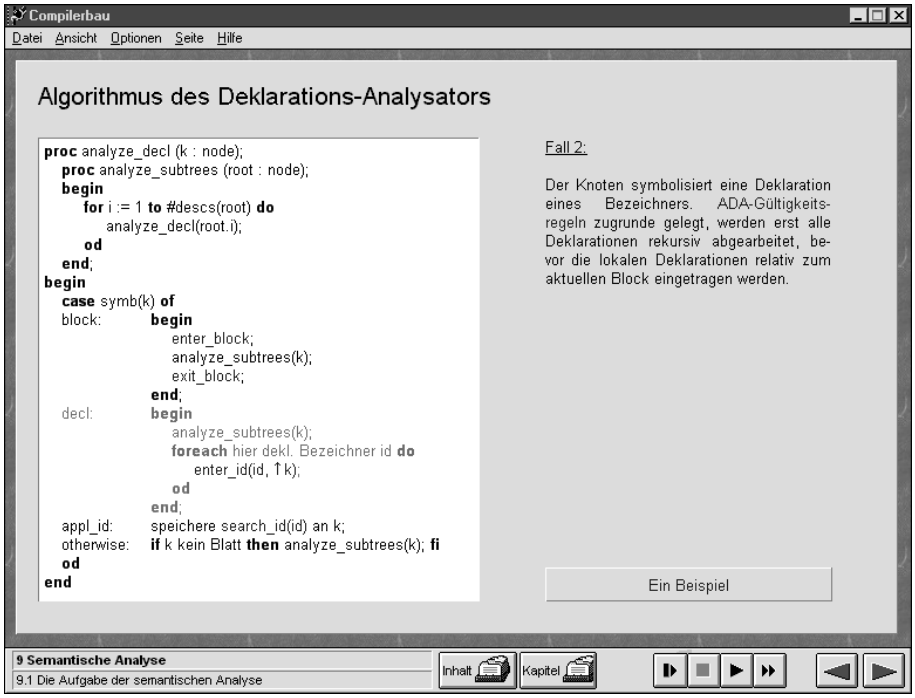


Abbildung 6.2: Statische Animation eines Deklarationsanalysators (AdsA).

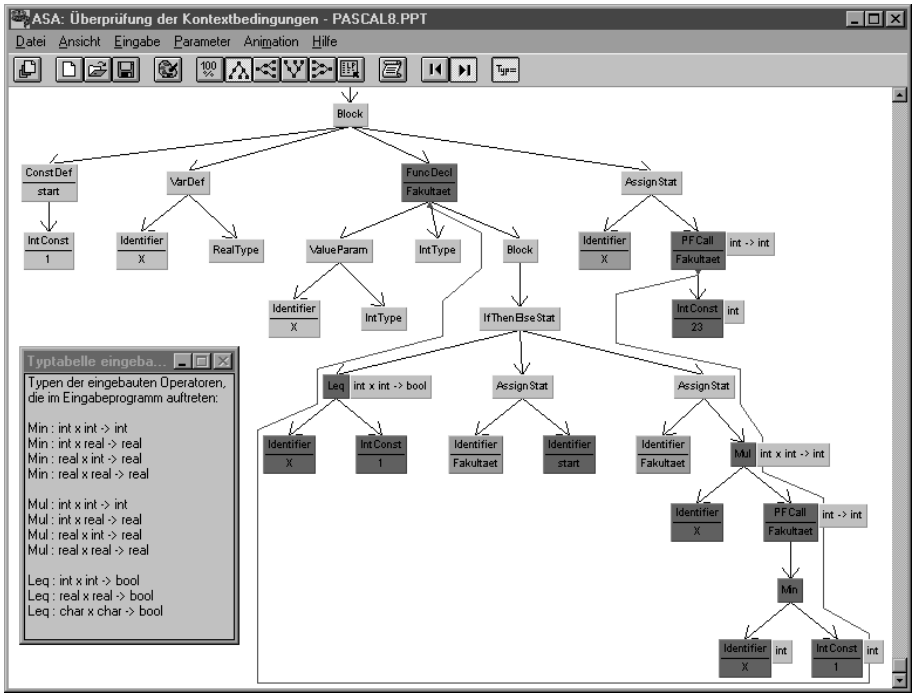


Abbildung 6.3: Visualisierung der Überprüfung der Kontextbedingungen (AdsA).

Dynamische Animationen Nun kommen wir zu den Visualisierungen, in denen der Anwender ein beliebiges Beispielprogramm oder eine beliebige Beispielspezifikation eingeben kann. Das Layout des resultierenden Syntaxbaums wird automatisch berechnet und im Anwendungsbereich angezeigt. Der Benutzer hat Einfluß auf das Baumlayout: Er kann die Abstände von Geschwisterknoten, benachbarten Knoten und Eltern/Kind-Knoten verändern. Weiterhin besteht die Möglichkeit, den Baum in seiner Größe zu verändern und in vier Richtungen zu orientieren. Diese Einflußmöglichkeiten sind notwendig, um den Baum optimal im Fenster zu plazieren.

Der Bildschirm in Abbildung 6.3 zeigt die Visualisierung einer Überprüfung der Kontextbedingungen in einem benutzerdefinierten Beispielprogramm. Eine komfortable Eingabemöglichkeit dieser Programme ist der eingebaute Editor, der auch eine Funktion zum Syntaxtesten bereithält. Ist das Eingabeprogramm syntaktisch fehlerhaft, so markiert der Editor das Fehlersymptom. Syntaktische Korrektheit ist eine Voraussetzung zur semantischen Analyse und wird daher vom System getestet. Der AST ist in der Abbildung fast vollständig dargestellt. Zu einigen Syntaxbaumknoten sind die Typattribute zu sehen. Grundlage für deren Berechnung sind die in einem Hilfsfenster (links unten) angegebenen Typen für die eingebauten Operatoren, die im Beispielprogramm verwendet wurden. In diesem Beispiel wurde für ein angewandtes Vorkommen des Bezeichners `Fakultaet` das nach den Gültigkeits- und Sichtbarkeitsregeln errechnete definierende Vorkommen nach einem Mausklick auf das angewandte Vorkommen farblich markiert und der entsprechende Link mit einer Kante gleicher Farbe symbolisiert. Wird kein definierendes Vorkommen gefunden, dann öffnet sich ein Dialogfenster mit einer entsprechenden Fehlermeldung. Der Benutzer hat nun die Möglichkeit, das Beispielprogramm abzuändern und die Analyse neu ablaufen zu lassen.

6.1.3 Generierung interaktiver Animationen von abstrakten Maschinen

Bei der Übersetzung höherer Programmiersprachen werden oftmals *abstrakte Maschinen* als plattformunabhängige Zwischenarchitekturen verwendet. Sie abstrahieren von den jeweiligen Besonderheiten realer Rechnerarchitekturen und sind für die Übersetzung einer bestimmten Quellsprache oder für Quellsprachen desselben Sprachparadigmas (imperativ, funktional, logisch, objektorientiert) besonders angepaßt. Dadurch vereinfachen sie die Implementierung von Übersetzern. Eine solche Implementierung generiert in einem ersten Schritt Code für die abstrakte Maschine. Anschließend kann dieser Code interpretiert oder in realen Maschinencode weitertransformiert werden. Durch die Unterteilung in zwei Schritte vereinfachen abstrakte Maschinen die Portierbarkeit und Wartbarkeit von Übersetzern. Ferner haben sie auch Vorteile, wenn der Übersetzerbau als Lerngebiet vermittelt werden soll: Lernende können sich auf die wichtigen Konzepte und Prinzipien konzentrieren und brauchen sich nicht um die speziellen Gegebenheiten realer Hardware zu kümmern, die sich permanent verändern.

In Abbildung 6.4 ist eine interaktive Animation einer abstrakten Maschine für eine funktionale Sprache zu sehen. An Programmspeicheradresse 19 auf der linken Seite der

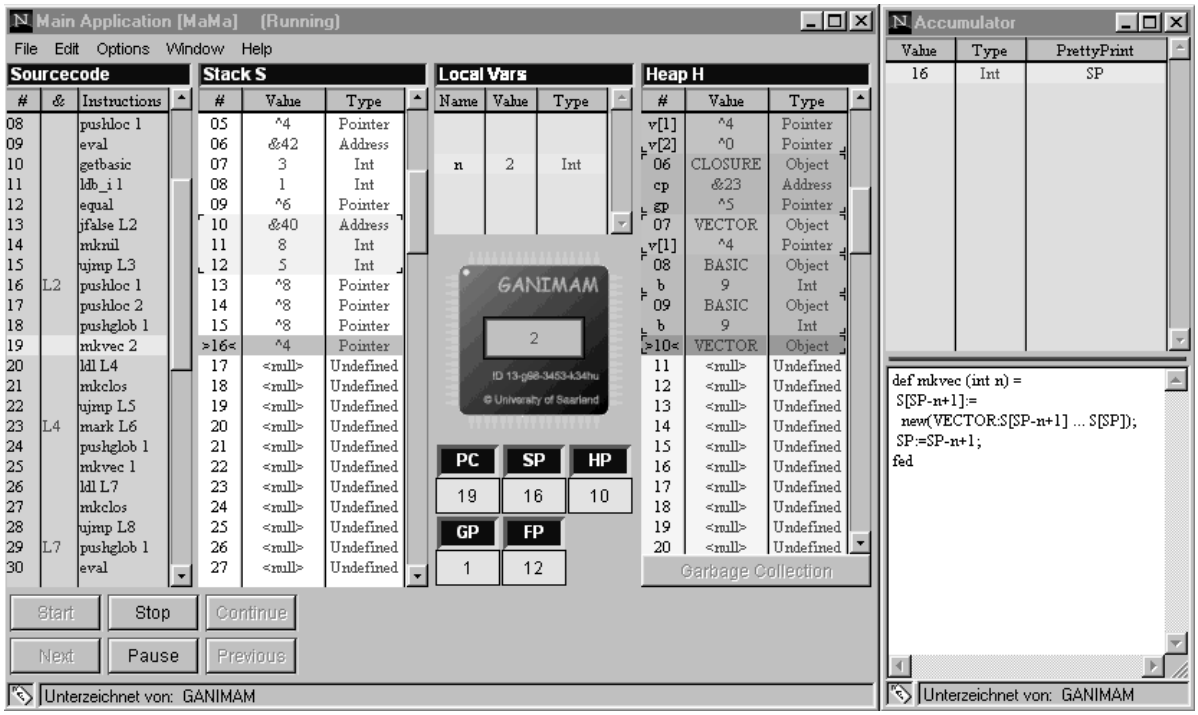


Abbildung 6.4: Momentaufnahme einer animierten abstrakten Maschine (GANIMAM).

Abbildung wird die Instruktion `mkvec 2` abgearbeitet. In Abhängigkeit von deren Definition, die in einem zusätzlichen Fenster angezeigt wird, kann der Benutzer den jeweils aktuellen Zustand der Maschine und den Prozeß der Instruktionsabarbeitung genau verfolgen. Beispielsweise ist zu sehen, daß auf der zweitobersten Kellerzelle an Kelleradresse 15 ein Zeiger mit der Haldenadresse 8 liegt, welcher auf das komplexe Haldenobjekt `BASIC` mit einer Komponente `b` verweist.

Als Beispiel für den in Abschnitt 1.1 auf Seite 2 beschriebenen ersten Ansatz zur Visualisierung von Generatoren betrachten wir hier das GANIMAM-System, einen webbasierten **G**enerator für interaktive **Anim**ationen **a**bstrakter **M**aschinen [DK98, Kun99, DK00c]. Dieses System, insbesondere sein graphisches Basispaket, stellte eine wichtige Erfahrungsgrundlage für die Gestaltung des GANIMAL-Frameworks dar. Im folgenden beschreiben wir, wie GANIMAM verwendet werden kann und was das System generiert.

Technischer Überblick GANIMAM kann über die Webseite des GANIMAL-Projekts [Gan02b] aufgerufen werden. Nachdem die Klassen des Applets vollständig geladen worden sind, stehen dem Benutzer mehrere bereits vorgegebene abstrakte Maschinen zur Auswahl. Die auf dem Server gespeicherten JAVA-Klassen der ausgewählten Maschine werden vom Applet geladen, und der Benutzer erhält eine vorinitialisierte abstrakte Maschinenvisualisierung, wobei z. B. die Register auf Defaultwerte eingestellt sind. Gegenwärtig existieren vordefinierte abstrakte Maschinen zu den Programmiersprachen Pascal, LaMa, C, Fun und Prolog. Zu den drei letztgenannten Sprachen sind auch die

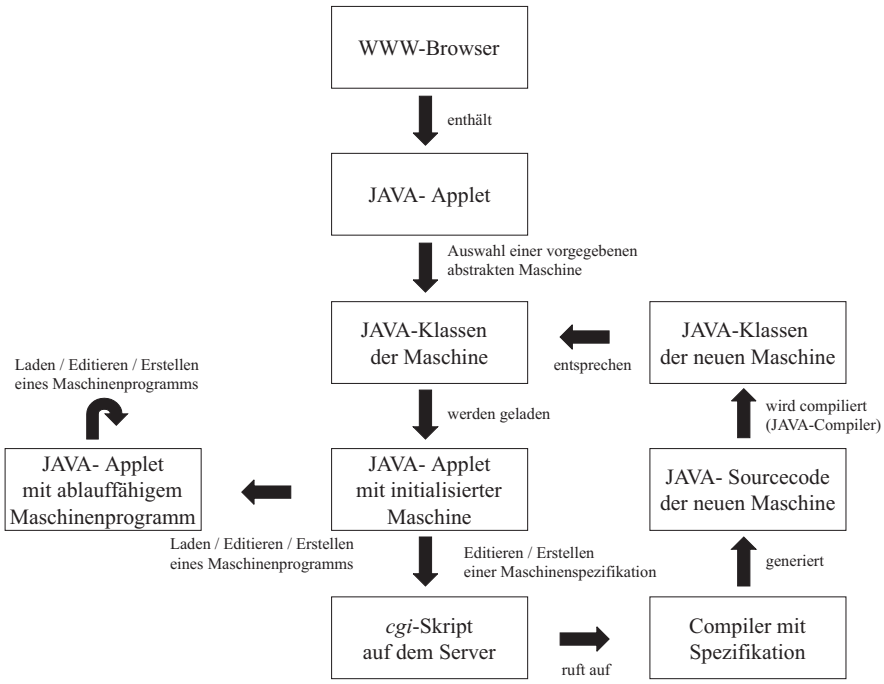


Abbildung 6.5: Interaktion der Systemkomponenten von GANIMAM (entnommen aus [Kun99]).

entsprechenden Codeerzeuger[1] auf dem Server installiert.

Nun hat man einerseits die Möglichkeit, bereits erstellte Maschinenprogramme für diese abstrakte Maschine zu laden und sich den Programmablauf animiert darstellen zu lassen. Weiterhin lassen sich selbsterstellte Maschinenprogramme eingeben, die vordefinierten Maschinenprogramme modifizieren, das Layout der verschiedenen Teile der visualisierten abstrakten Maschine anpassen und die animierte Ausführung eines Maschinenprogramms steuern. Andererseits hat der Benutzer nun die Option, die bereits geladene Maschine selbst zu ändern, indem er die entsprechende Spezifikation modifiziert oder ganz neu eingibt. Das Applet schickt diese neue Spezifikation zum Server. Ein CGI-Skript auf dem Server erzeugt JAVA-Quellcode, der durch einen JAVA-Compiler in Klassendateien der neuen Maschine übersetzt wird. In Kombination mit dem GANIMAM-Basispaket bilden diese Klassendateien ein interaktives JAVA-Applet. Nachdem das Applet diese Klassen per JAVA-Reflection über das Internet geladen hat, kann der Benutzer für diese neue Maschine wieder neue Maschinenprogramme eingeben und deren Ablauf visualisieren oder erneut die abstrakte Maschinenspezifikation ändern und sich auf dem Server eine neue abstrakte Maschine erzeugen lassen usw. (siehe Abb. 6.5).

Da abstrakte Maschinen eine unterschiedliche Anzahl von Registern, Kellern und Halden haben können, wird für jede generierte Maschine ein automatisches Layout benötigt. Dieses ordnet die verschiedenen Speicherarten um den Akkumulator, eine konzeptionelle

[1] Diese Codeerzeuger werden mit dem Applet zusammen im Rahmen des ULI-Projekts [ULI02] gepflegt und weiterentwickelt.

Recheneinheit, die als Chip in der Mitte von Abbildung 6.4 zu sehen ist, herum an. Mit dem Akkumulator ist ein Akkumulatorfenster verbunden, das den Ausdruck anzeigt, der gerade im Akkumulator berechnet wird. Das Fenster stellt außerdem die Definition der aktuell ausgeführten Instruktion bzw. Funktion dar. Während der Ausführung einer Instruktion zeigt eine Animation den Informationsfluß von Registern oder Speicherzellen in den Akkumulator und zurück. Über Mausklicks auf die verschiedenen Komponenten können z. B. Instruktionsdefinitionen in das Akkumulatorfenster geladen oder die Werte von Registern neu gesetzt werden.

Spezifikationssprache Für die Spezifikation von selbstdefinierten abstrakten Maschinen wird eine Spezifikationssprache [DKP97] verwendet, die auf einer Notation von Wilhelm und Maurer [WM96] basiert. Kern dieser Sprache ist eine Kontrollflußsprache mit Zuweisungen, Ausdrücken, Bedingungen und Schleifen. Sie setzt auf einem allgemeinen Maschinenmodell auf, bestehend aus einer Menge von Instruktionen, einem Programmspeicher, Halden, Kellern und Registern. Die Maschine läuft in einer Schleife, welche genau die Instruktion ausführt, auf die der Wert eines speziellen Registers (Programmzähler PC) zeigt. Innerhalb dieses Modells kann eine Maschinenspezifikation durch die Deklaration ihrer Keller, Halden und Register sowie der Definition ihrer Instruktionen formuliert werden (nach [DK00c]). Die Sprache erleichtert den Spezifikationsprozeß dadurch, daß der Spezifizierer auch komplexe Hilfsfunktionen deklarieren kann.

```
STACK S[100] with SP=-1;        def mkvec (int n) =
HEAP H[100]  with HP=-1;          S[SP-n+1] :=
REGISTER PC, FP, GP;                new(VECTOR:S[SP-n+1] ... S[SP]);
                                  SP := SP-n+1;
OBJECT                          fed
  FUNVAL  (cf, fap, fgp),
  BASIC   (b),                  def greater =
  CLOSURE (cp,gp),                if(S[SP-1] > S[SP]) then
  VECTOR[](v);                      S[SP-1] := true;
                                  else
                                    S[SP-1] := false;
                                  fi;
                                  SP := SP-1;
                                fed
```

Abbildung 6.6: Auszug aus einer Maschinenspezifikation.

Abbildung 6.6 zeigt ein Fragment einer Spezifikation einer abstrakten Maschine für die funktionale Sprache LaMa aus [WM96]. Es werden ein Keller `S` mit Kellerzeiger `SP`, eine Halde `H` mit Haldenzeiger `HP` und drei weitere Register deklariert. Keller- als auch Haldenzeiger verweisen auf die jeweils oberste Speicherzelle. Danach führt die Spezifikation vier verschiedene komplexe Haldenobjekte zur Konstruktion strukturierter Datentypen ein. Die nachstehenden Elemente in runden Klammern bezeichnen ihre jeweiligen Kom-

ponenten. Anschließend werden die beiden Instruktionen `mkvec` und `greater` definiert. Die Instruktion `mkvec` erzeugt beispielsweise ein neues Objekt des Datentyps `VECTOR`, speichert ein Feld von Werten aus dem Keller darin ab und hinterlegt die neue Objektreferenz wieder auf dem Keller.

Nach der Generierung der Implementierung der abstrakten Maschine kann der Benutzer Programme in der Sprache der neu spezifizierten abstrakten Maschine eingeben, diese Schritt für Schritt ausführen und den Inhalt eines jeden Registers und jeder Speicherzelle inspizieren.

Während der Animation einer abstrakten Maschine ist es wichtig, daß man nicht nur Daten und Vorgänge auf dem geringen Abstraktionsniveau und der Granularität der Spezifikationssprache darstellt, sondern auch Abstraktionen auf höherem Niveau. Um dies zu ermöglichen, wurden einerseits Visualisierungsannotationen zur Spezifikationssprache hinzugefügt. Diese Annotationen ähneln den Interesting Events einschlägiger Algorithmenanimationssysteme, wie sie in Kapitel 2 beschrieben wurden. Andererseits erlaubt das System, die Animation atomarer Berechnungen, die im Rumpf von Instruktionsdefinitionen ausgeführt werden, über einen Mausklick abzuschalten.

6.2 Animation des Heapsort-Algorithmus

Die Animation von Sortieralgorithmen hat im Forschungsbereich der Softwarevisualisierung eine lange Tradition. Neue Systeme und Ansätze werden häufig anhand von Sortierverfahren ausgetestet. Ferner nutzen Entwickler von Algorithmenanimationssystemen diese Verfahren als Anwendungsfälle, um die Eigenschaften ihrer Systeme einem breiten Publikum, etwa auf Konferenzen oder Workshops, vorzustellen. Gründe dafür liegen in der Überschaubarkeit der Algorithmen und der oftmals natürlichen, graphischen Repräsentation der Zustände und Datenstrukturen. Dieser Abschnitt zeigt Schritt für Schritt die Vorgehensweise auf, wie der Visualisierer eine Animation des Heapsort-Algorithmus in Ganila erstellt, und wie sich die erzeugte Animation einem späteren Benutzer präsentiert.

Heapsort arbeitet in zwei Phasen: Zuerst strukturiert der Algorithmus die zu sortierenden Elemente zu einem *Heap*, einem vollständigen binären Baum, in dem der Wert eines jeden Knotens größer ist als die einzelnen Werte seiner Kinder. Danach wird die Wurzel (d. h. der Knoten mit dem größten Wert) des Heaps gelöscht, an eine (sortierte) Liste angehängt und die Heapeigenschaft wiederhergestellt. Dies geschieht solange, bis die Heapstruktur leer ist. Heaps können als Felder implementiert werden. Hierbei wird der Wurzel die Position 0 zugewiesen, sowie für jeden Knoten an Feldindex i dessen linkes Kind an Position $2i + 1$ und dessen rechtes Kind an Position $2i + 2$ plaziert.

6.2.1 Implementierungssicht

Die Produktion einer Animation in Ganimal erfolgt analog zur Vorgehensweise in klassischen, eventgetriebenen Animationssystemen wie Zeus oder Polka.

Konzeptioneller Entwurf Zunächst muß sich der Entwickler darüber klar sein, wie die Animation aussehen soll. In unserem Fall möchten wir die Heapstruktur als Feld (`ArrayView`) und als Baum (`TreeView`) darstellen. Sowohl die Feldelemente als auch die Knoten des Baums sollen die zu sortierenden Elemente als Zahleneintrag enthalten. Ungünstigerweise visualisieren diese Sichten auf den Algorithmus nur die Elementposition in der Datenstruktur. Der optische Vergleich zweier Elemente ist lediglich über den jeweiligen Zahlenwert möglich. Eine Lösung dieses Problems bietet die sogenannte Balkensicht (`Stick-` oder `BarView`), in der Werte als Balken unterschiedlicher Höhe repräsentiert sind. Für einen gewissen Zeitraum bietet sich zu Testzwecken zusätzlich eine Sicht auf die abgearbeiteten IEs (EventView, siehe S. 91) an.

Identifizierung der Interesting Events Als nächstes muß der Quellcode mit IEs annotiert werden. Für den Heapsort-Algorithmus genügen die folgenden vier Events:

- *MarkNodes(Knoten 1, Knoten 2, Feldreferenz).* Zur Herstellung der Heapeigenschaft werden zwei Knoten miteinander verglichen. Beide erhalten zunächst eine identische Markierung.
- *MarkMax(Knoten 1, Knoten 2, Feldreferenz).* Der erste Knoten ist größer als der zweite Knoten und wird farblich hervorgehoben. Nach einigen Momenten erhalten beide Knoten wieder die ursprüngliche Farbe.
- *Exchange(Knoten 1, Knoten 2, Feldreferenz).* Die Heapeigenschaft zwischen beiden Knoten ist verletzt, sie müssen vertauscht werden. Dieser Vorgang verläuft je nach Sicht unterschiedlich, z. B. sollen in der Balkensicht zwei Balken animiert an den jeweils anderen Platz verschoben werden.
- *DisableNode(Knoten, Feldreferenz).* Die Heapeigenschaft ist erfüllt, d. h. an der Wurzel befindet sich das aktuelle Maximum. In der korrespondierenden Animation wird der Wert an die letzte Stelle des Feldes verschoben und vom Heap durch Dekrementierung der Heapgröße abgekoppelt.

Exemplarisch wird hier nur die Methode `exchange()` des Heapsort-Algorithmus ausführlich beschrieben. In Anhang A sind der vollständige GANILA-Quellcode sowie die

```
public void exchange(int i, int j) {
  int help;

  help = A[i];
  A[i] = A[j];
  A[j] = help;
  *IE_Exchange(i, j, A);
}
```

Programm 6.1: Definition I der Methode `exchange()`.

JAVA-Klassen einer möglichst einfachen Implementierung der Balkensicht abgedruckt. Die Methode `exchange()` vertauscht zwei Feldelemente des global definierten Eingabefeldes `A`; dazu ist die Verwendung einer Hilfsvariablen `help` notwendig. Programmausschnitt 6.1 zeigt die Definition dieser Methode und ein bereits eingefügtes Interesting Event `*IE_Exchange()`.

Erreicht die Ausführung dieses IE, dann ruft das GANIMAL-Framework gleichzeitig die entsprechenden Eventhandler aller angemeldeten Sichten (s. u.) auf. Die Visualisierung dieses IE hängt nun von der Implementierung der Sichten ab. Geht man davon aus, daß der Entwickler den Gebrauch der Hilfsvariablen `help` nicht in irgendeiner Weise innerhalb einer Sicht simulieren möchte, dann wird dieser Aspekt in der Animation nicht dargestellt. Es ist jedoch oft erwünscht, von solchen sehr konkreten Details zu abstrahieren. GANILA bietet mit dem `*FOLD`-Operator die Möglichkeit, dem Anwender diese Entscheidung zur Laufzeit zu überlassen.

```
public void exchange(int i, int j) {
  int help;

  *{ *IE_MoveToTemporary(i, A);
      help = A[i];
      *BREAK;
      *IE_MoveElement(i, j, A);
      A[i] = A[j];
      *IE_MoveFromTemporary(j, A);
      A[j] = help;
  *} *FOLD *{ *IE_Exchange(i, j, A); *}
}
```

Programm 6.2: Definition II der Methode `exchange()`.

Hierzu sind, wie im Programmbeispiel 6.2 gezeigt, drei neue IEs einzubauen. Das erste Event animiert den Fluß des Wertes von dem Knoten mit Index `i` zu der Hilfsvariablen. Danach soll die Animation solange anhalten, bis der Anwender weitermachen möchte. Die Wertzuweisung von Feldelement `j` nach Feldelement `i` wird durch das zweite IE visualisiert. Zuletzt wird das Zurückschreiben aus der Hilfsvariablen zum Element `j` dargestellt. Erscheint diese Vorgehensweise später zu feinkörnig, so ist eine Umschaltung auf das Event `*IE_Exchange()` zur Laufzeit möglich, wenn sich die Animation nicht gerade im Gültigkeitsbereich des `*FOLD`-Operators befindet. Dabei werden die IEs innerhalb des ersten Klammerpaares ignoriert, der `*BREAK`-Operator und alle sonstigen Zuweisungen allerdings nicht. Zu beachten ist, daß in diesem Fall der direkte Werteaustausch durch die Implementierung des Eventhandlers von `*IE_Exchange()` wahrscheinlich parallel visualisiert werden würde. In analoger Weise ist für den restlichen Quellcode des Algorithmus zu verfahren und schließlich der Compiler GAJA aufzurufen, der u. a. eine Templateklasse `HeapsortView` generiert, von der alle selbstdefinierten Sichten erben müssen.

Design der Sichten Im nächsten Schritt sind die Sichten und die Eventhandler der IEs zu implementieren. Der Compiler GAJA vereinfacht deren Implementierung, indem er alle Eventhandler mit leeren Methodenrümpfen bereits in der Klasse `HeapsortView` erzeugt. Diese sind bei Bedarf einfach zu überschreiben. Als Beispiel betrachten wir den Ausschnitt einer sehr leicht zu implementierenden Sicht in Programm 6.3: die Balkensicht bzw. `BarView`.

```
public class BarView extends HeapsortView {
  GBarRenderer barRenderer;

  public BarView(Gint[] array) {
    setName("Bar View");
    barRenderer = new GBarRenderer(array, GBarRenderer.BAR3D_MODE, true);
    addRenderer(barRenderer);
  }

  public void IE_Exchange_Play_Visible(Gint i, Gint j, Gint[] A) {
    barRenderer.invalidate();
  }
  ...
}
```

Programm 6.3: Implementierung der Sicht `BarView`

Die `BarView` verwendet für die Balkendarstellung eine Klasse `GBarRenderer` des graphischen Basispakets. Der Klassenkonstruktor der Sicht initialisiert diese Darstellung mit dem Eingabefeld des Heapsort-Algorithmus und mit einem Flag für die 3D-Ansicht der Balken, d. h. die Animation startet mit einem bereits mit den Eingabewerten vorinitialisierten 3D-Balkendiagramm. Da in unserem Beispiel mit der Referenz des Eingabefeldes gearbeitet wurde, reduziert sich die Implementierung des Eventhandlers für das Interesting Event `*IE_Exchange()` auf ein Neuzeichnen der Balkendarstellung im sichtbaren *PLAY*-Modus (vgl. Abschnitt 3.2.4). Der Grund liegt darin, daß die Sicht im Sinne der Animation keinen eigenen Zustand besitzt. Die Algorithmenausführung ändert das Feld `A` und somit über die Referenz auch die Balkendarstellung. Falls das Event deaktiviert ist, werden die Balkengraphiken einfach nicht mehr aktualisiert. Folglich ist die Deaktivierung eines einzelnen Events eher bei in sich abgeschlossenen Ereignissen (z. B. bei einem auditiven Signal) sinnvoll. Wird ein Event aus einer Reihe von Events deaktiviert, die voneinander abhängige Visualisierungen auslösen, kann die verbleibende Visualisierung den Betrachter möglicherweise irritieren.

Ein wichtiges Bindeglied fehlt in diesem Beispiel noch. Nachdem die Entwicklung aller Sichten beendet worden ist, müssen diese noch registriert werden, damit das GANIMAL-Laufzeitsystem Events zu ihnen schicken kann. Weiterhin hängen die Sichten sozusagen „in der Luft“, sie müssen noch innerhalb eines oder mehrerer Fenster positioniert werden. Das nachstehende Programmcodefragment 6.4 erzeugt ein Fenster der Größe 600×500 Pixel und ordnet die `Tree`- und `BarView` nebeneinander in der oberen Fensterhälfte sowie

die `ArrayView` im unteren Fensterbereich an, wie in Abbildung 6.8 zu sehen. Dabei ist die EventView in die `ArrayView` eingebaut und wird von dieser auch verwaltet.

```
public class Heapsort extends GFrame {
  BarView bv;

  public Heapsort(GAlgorithm alg) {
    super("Heapsort");
    setSize(600, 500);

    GPanel gp = new GPanel();
    gp.setLayout(new GridLayout(2, 1));
    GPanel top = new GPanel(new GridLayout(1, 2));
    ...
    bv = new BarView(((HeapsortAlgorithm)alg).A);
    top.add(bv);
    gp.add(top);
    add(gp);
    ...
    GEventControl control = alg.getControl();
    control.addIEListener(bv);
    ...
  }
}
```

Programm 6.4: Registrierung der Sichten.

Der Programmcode am Ende des Fragments registriert die `BarView` bei der Visualisierungskontrolle. Nach erfolgreicher Übersetzung durch den Java-Compiler erhält man eine Animation, die sowohl als eigenständige Anwendung verwendbar ist als auch in ein hypermediales Lernsystem eingebettet werden kann.

6.2.2 Anwendungssicht

Nachdem der Anwender den Heapsort-Algorithmus über eine Dialogbox geöffnet hat, wird der AST des Eingabeprogramms in die graphische Benutzerschnittstelle geladen. Abbildung 6.7 zeigt diese Schnittstelle[2], die in Abschnitt 5.2 ausführlich beschrieben ist. Ein Teil der Unterbäume des AST wurde der Übersichtlichkeit halber ausgeblendet, nur die Methode `exchange()` ist vollständig sichtbar. Man kann die einzelnen Knoten über die Maus auswählen und kontextabhängige Eigenschaften setzen. Hier wurde beispielsweise das Event `*IE_Exchange()` angeklickt. Der Benutzer kann an diesem Programmpunkt einen Haltepunkt setzen oder das IE deaktivieren. Über diesen Weg kann

[2] Abb. 6.7 ist mit Abb. 5.2 in Abschnitt 5.2 identisch. Sie wurde hier erneut abgedruckt, damit sich der Leser eine bessere Vorstellung darüber machen kann, wie das Zusammenspiel zwischen der Steuerung über die GUI und der Animation für einen bestimmten Algorithmus aussehen kann.

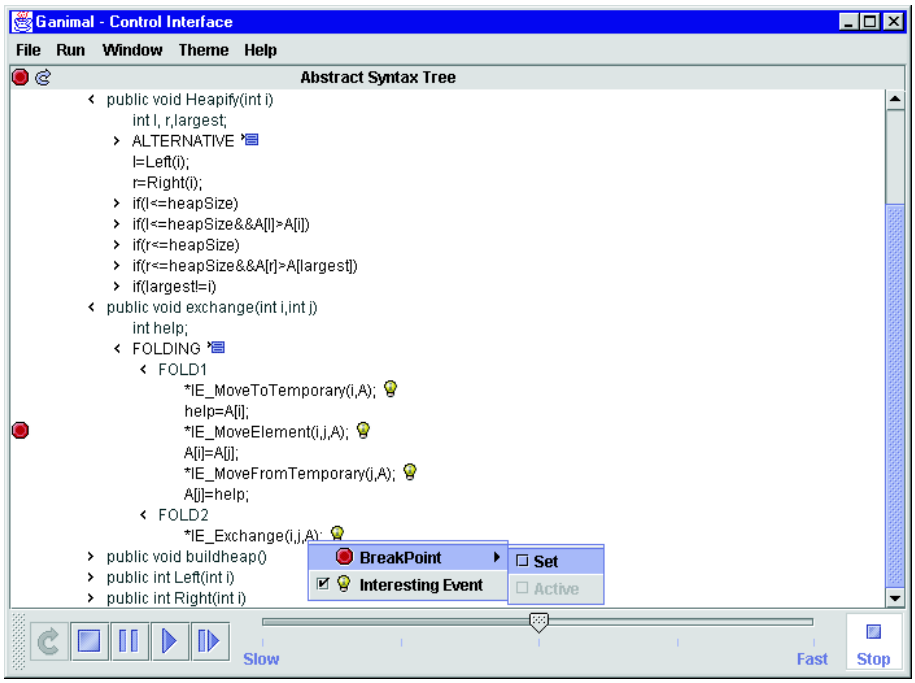


Abbildung 6.7: GUI mit AST des Heapsort-Algorithmus.

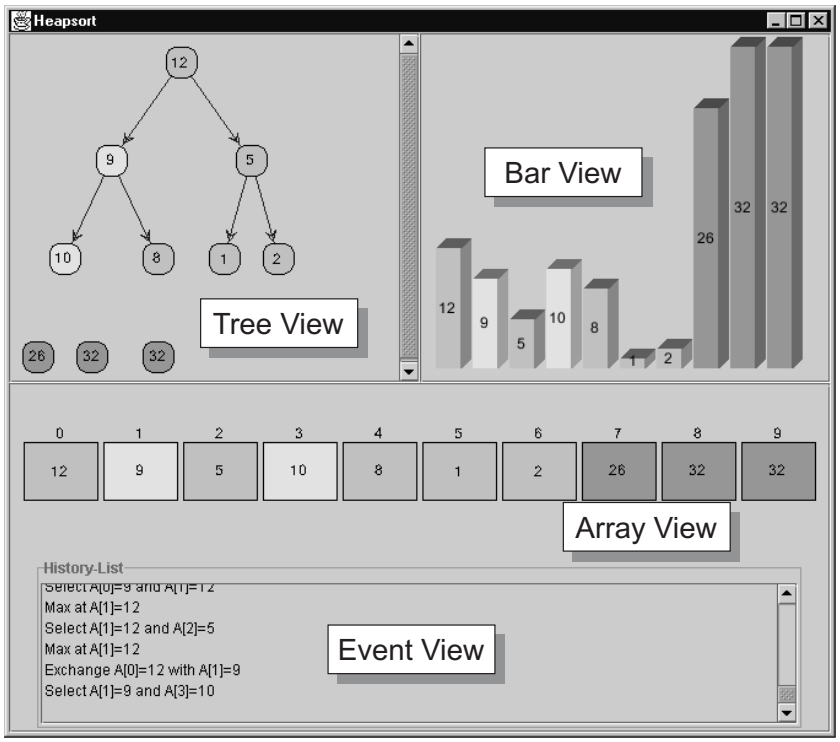


Abbildung 6.8: GANIMAL-Algorithmenanimation von Heapsort.

er die Animation vor ihrem Start oder während der Ausführung seinen Bedürfnissen entsprechend anpassen.

Abbildung 6.8 zeigt eine Bildschirmaufnahme des laufenden Algorithmus. Die oben beschriebenen Sichten sind in einem Fenster angeordnet, und im aktuellen Schritt werden die Knoten mit den Werten 9 und 10 farblich markiert. Wie man in der Darstellung erkennt, ist die Heapeigenschaft zwischen den beiden Knoten verletzt; sie werden demzufolge im nächsten Animationsschritt miteinander verglichen, der Knoten mit dem größeren Wert wird hervorgehoben und anschließend werden sie vertauscht.

6.3 Animation der Generierung endlicher Automaten

In Abschnitt 6.1.1 stellten wir eine Lernsoftware über die lexikalische Analyse [BDKW99] vor. Ihre Animationen zeigen sowohl wie endliche Automaten aus regulären Ausdrücken erzeugt werden als auch wie endliche Automaten arbeiten. Der Benutzer kann allerdings die regulären Ausdrücke und Eingabeworte der deterministischen bzw. nichtdeterministischen Automaten in diesen Animationen nicht verändern.

Eine komplexere Beispielanwendung des GANIMAL-Frameworks[3] ist das System GANIFA, das aus einem HTML-basierten, interaktiven Textbuch über die Automatentheorie sowie einem Applet zur Visualisierung und Animation von Algorithmen aus diesem Bereich besteht. Dieses GANIFA-Applet kann von der Webseite [Gan02a] als JAR-Datei heruntergeladen werden und benötigt ein aktuelles JAVA2-Plugin. Lehrende können es für ihre eigenen webbasierten Übungen, Vorlesungsskripte oder Präsentationen von endlichen Automaten nutzen. Weiterhin gibt diese Webseite eine kurze Übersicht, wie das Applet an die eigenen Bedürfnisse angepaßt und in HTML-Seiten eingebaut wird.

6.3.1 Elektronisches Textbuch

Das GANIFA-Applet wurde in ein elektronisches Textbuch [BDK+02] über Automatentheorie eingebettet, welches mit Hilfe eines üblichen Webbrowsers wie Netscape Communicator oder Microsoft Internet Explorer studiert werden kann. Im Augenblick gibt es sowohl eine englische als auch eine deutsche Version des Textbuchs und des Applets selbst.

Als eine Einführung in die Theorie der endlichen Automaten gibt das Textbuch eine Übersicht über formale Sprachen im allgemeinen, über reguläre Sprachen sowie über reguläre Ausdrücke. Anschließend werden Übergangsdiagramme, nichtdeterministische und deterministische endliche Automaten beschrieben. Formale Definitionen werden in einem separaten Hilfsfenster des Browsers (siehe Abb. 6.9 auf S. 119) angezeigt, wenn der Benutzer auf den entsprechenden Hyperlink innerhalb der HTML-Seite klickt. Man kann dieses Fenster geöffnet lassen und jederzeit eine andere Definition aufrufen oder

[3] Aufgrund des chronologischen Ablaufs der Entwicklung von GANIFA-Applet und GANIMAL-Framework nutzt das GANIFA-Applet lediglich die GANIMAL-Laufzeitumgebung, jedoch noch nicht die Vorteile der Reifikation von Programmpunkten durch die aktuelle Version des Compilers GAJA.

das Fenster über einen Knopf schließen. Das Textbuch präsentiert vier Algorithmenanimationen, die in Abschnitt 6.3.2 beschrieben sind und mit Hilfe des Applets dargestellt werden.

Beispielsweise zeigt Abbildung 6.10 eine Lektion über die Minimierung eines deterministischen endlichen Automaten (kurz DEA). Im unteren Rahmen der dargestellten HTML-Seite ist eine knappe Motivation und Beschreibung des Minimierungsalgorithmus *DEA* → *minDEA* zu sehen. Von dieser Seite gelangt der Benutzer über einen Hyperlink zu einer weiteren Seite, auf der er einen regulären Ausdruck angeben und die Animation des Algorithmus durch Betätigung einer Schaltfläche starten kann. Diese Seite enthält zudem eine Legende über die Ereignisse des Startvorgangs und einige Zeichenerklärungen für die Animationen.

Alle Seiten des elektronischen Textbuchs sind oben und unten mit einer Navigationsleiste versehen. Mit ihrer Hilfe kann man blättern und auf die Titelseite zurückspringen. Auf der Titelseite ist ein Inhaltsverzeichnis mit Direktzugriff auf die einzelnen Kapitel angelegt. In der Navigationsleiste befindet sich zusätzlich noch ein Popup-Menü, mit Hilfe dessen man direkt zu allen anderen Kapiteln wechseln kann. Der voreingestellte Wert dieses Popup-Menüs ist das aktuell angezeigte Kapitel. Somit behält der Benutzer den Überblick, an welcher Stelle im Textbuch er sich gerade befindet. Desweiteren besteht hier die Möglichkeit, auf die englische (bzw. deutsche) Version des Textbuchs umzuschalten.

6.3.2 GANIFA-Applet

Das GaniFA-Applet visualisiert und animiert die folgenden Algorithmen und Simulationen in der Reihenfolge ihrer Auflistung:

1. Generierung eines nichtdeterministischen endlichen Automaten (NEA) aus einem regulären Ausdruck (RA) [WM96].
2. Beseitigung der ε-Übergänge eines NEA [RS59, WM96].
3. Transformation eines deterministischen endlichen Automaten (DEA) aus einem NEA ohne ε-Übergänge [RS59, WM96].
4. Minimierung eines deterministischen endlichen Automaten (minDEA) [HU79].

(5.) Das Applet kann die Berechnung eines jeden der oben generierten Automaten auf einem Eingabewort visualisieren.

Die Literaturangaben deuten auf Quellen, in denen der entsprechende Algorithmus detailliert erläutert und ggf. auch bewiesen wurde. Da es sich um klassische Verfahren handelt, die teilweise seit Jahrzehnten bekannt und Bestandteil eines jeden Informatikstudiums sind, werden sie in dieser Arbeit nicht näher beschrieben.

GaniFA erhält als Eingabe einen regulären Ausdruck oder die Beschreibung eines endlichen Automaten. Die Eingabe des endlichen Automaten ist dabei nur über die

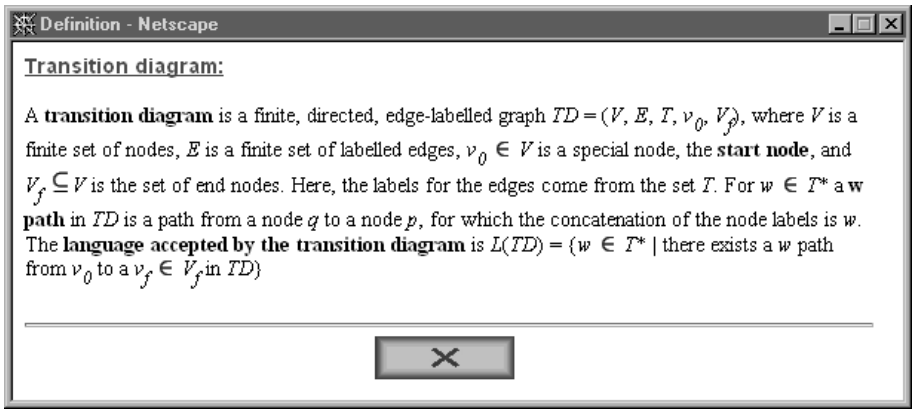


Abbildung 6.9: Definitionenfenster.

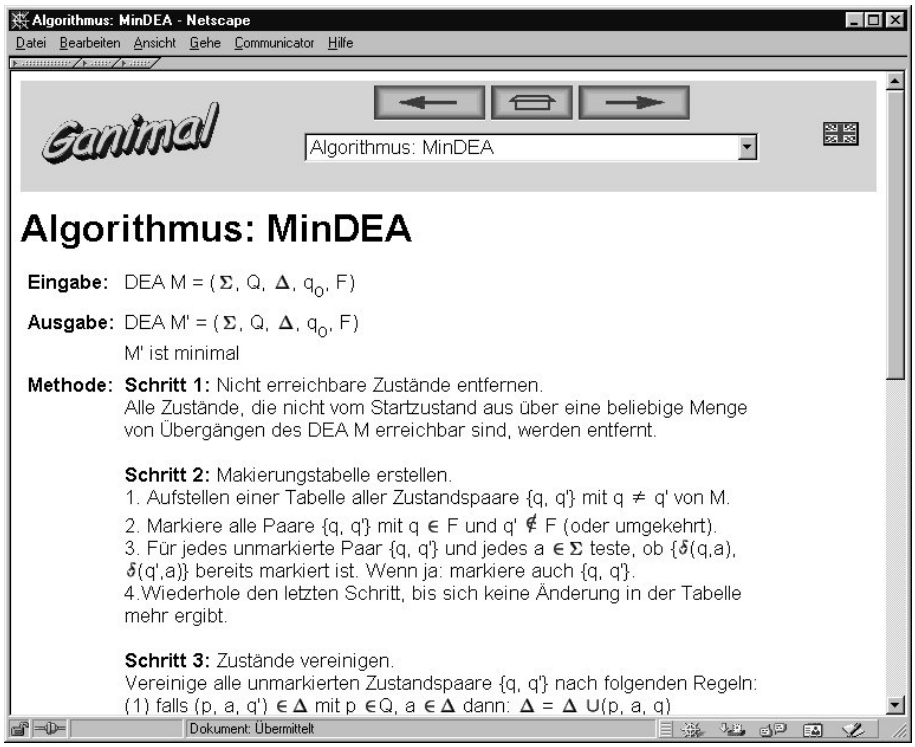


Abbildung 6.10: Bildschirmaufnahme des elektronischen Textbuchs.

Appletparameter (siehe Abschnitt 6.3.2.2) möglich. Reguläre Ausdrücke können entweder über einen Appletparameter oder zur Laufzeit über ein Eingabefeld spezifiziert werden. Für die Simulation der endlichen Automaten kann zusätzlich ein Eingabewort übergeben werden. Alle Algorithmen und Simulationen (bis auf den ersten Algorithmus $RA \rightarrow NEA$) verwenden als „Eingabeautomat“ die Ausgabe des vorherigen Algorithmus und zwar unabhängig davon, ob die Animation über die Parameter eingeschaltet wurde oder nicht.

Der Benutzer kann den Zeitpunkt der Visualisierung des nächsten Animationsschritts selbst wählen (*Step*). Er kann aber auch alle Animationsschritte automatisch anzeigen lassen, wobei zwischen jeweils zwei Schritten kurze Pausen eingebaut sind (*Play*). Es besteht außerdem jederzeit die Möglichkeit, die Animation eines Algorithmus bzw. einer Simulation abzubrechen oder zwischen den beiden Steuerungszuständen *Play* und *Step* umzuschalten.

6.3.2.1 Animationen

Dieser Abschnitt erläutert beispielhaft die Animation von zwei Algorithmen: die Transformation eines DEA aus einem NEA ohne ε-Übergänge sowie die Minimierung eines DEA.

NEA ohne ε-Übergang $\rightarrow$ DEA Dem Verfahren liegt die Idee zugrunde, daß ein Zustand des DEA einer Menge von Zuständen des NEA entspricht (sog. *Teilmengenkonstruktion*). Der DEA merkt sich in seinen Zuständen alle möglichen Zustände, in denen sich der NEA nach dem Lesen der jeweiligen Eingabezeichen befinden kann, d. h. der DEA befindet sich nach dem Lesen der Eingabe $z_1 z_2 \ldots z_n$ in einem Zustand, der die Teilmenge aller Zustände des NEA repräsentiert, die vom Startzustand des NEA entlang eines mit $z_1 z_2 \ldots z_n$ gekennzeichneten Weges erreichbar sind. Zu Beginn besteht der DEA aus genau einem Zustand, der wegen der vorangegangenen Entfernung der ε-Übergänge dem Startzustand des NEA entspricht.

Bei der Animation dieses Algorithmus werden zwei Übergangsdiagramme angezeigt (siehe Abb. 6.11): Links ist die Eingabe des Algorithmus (NEA ohne ε-Übergänge) dargestellt. Dieses Übergangsdiagramm wird nur für die farbliche Markierung aller Knoten und Kanten benutzt, die zu diesem Zeitpunkt betrachtete Zustände und Übergänge repräsentieren. Der dargestellte NEA verändert sich in seiner Struktur während der Animation nicht. In der rechten Hälfte wird ein DEA dargestellt, soweit er schon vom Algorithmus erzeugt worden ist. Im letzten Animationsschritt erscheint an dieser Stelle die endgültige Ausgabe des Algorithmus.

DEA $\rightarrow$ minDEA Der Algorithmus zur Minimierung eines DEA sowie dessen Animation erfolgt in mehreren Phasen:

1. Anzeigen der Eingabe des Algorithmus (zu minimierender DEA).
2. Feststellen und Entfernen aller vom Startzustand aus nicht erreichbaren Zustände und Übergänge.

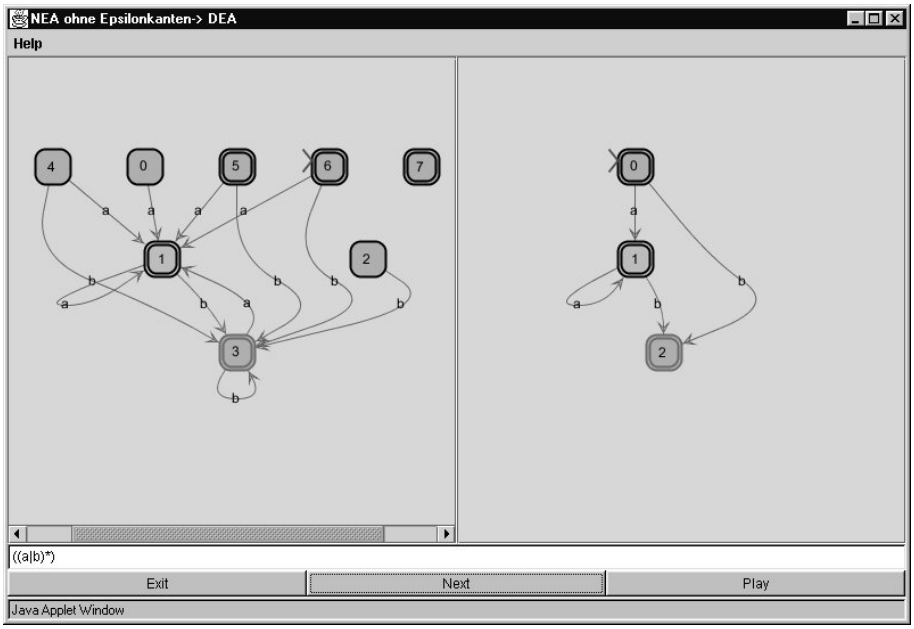


Abbildung 6.11: Animation des Algorithmus $NEA \rightarrow DEA$. Der NEA und der erzeugte DEA akzeptieren die Sprache $L((a|b)^*)$.

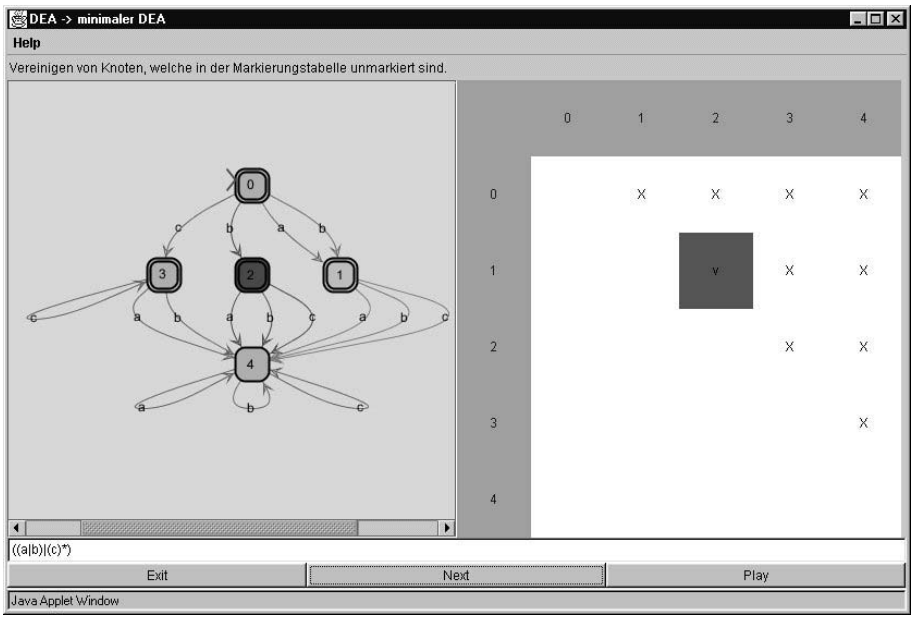


Abbildung 6.12: Animation des Algorithmus $DEA \rightarrow minDEA$. Der DEA und der erzeugte minimale DEA akzeptieren die Sprache $L(a|b|c^*)$.

3. Suchen von Zustandspaaren, die aus einem Endzustand und einem Nichtendzustand bestehen. Diese Paare dienen der Erstellung einer Markierungstabelle.

4. Markieren von Zustandspaaren, die auf markierte Zustandspaare zurückführbar sind.

5. Vereinigen von Zuständen anhand der Markierungstabelle.

6. Anzeigen der Ausgabe des Algorithmus (minimaler DEA).

Mit Abbildung 6.12 ist ein Ausschnitt einer Animation dieses Algorithmus gegeben. Mit Hilfe einer Textzeile am oberen Rand wird angegeben, in welcher Phase sich die Animation aktuell befindet (hier in Phase 5). In der linken Fensterhälfte wird ein Übergangsdiagramm des vom Algorithmus bis zum aktuellen Animationsschritt erzeugten endlichen Automaten dargestellt. Im ersten Animationsschritt entspricht dies dem Eingabeautomat und im letzten Schritt der Ausgabe (minimaler DEA) des Algorithmus. Rechts wird die vom Algorithmus benutzte Markierungstabelle angezeigt.

6.3.2.2 Adaptabilität

GaniFA kann über eine große Anzahl von Parametern an verschiedene Bedürfnisse angepaßt und auch leicht in Webseiten integriert werden. Es ist möglich, nur eine Auswahl der Algorithmen zu visualisieren und einen endlichen Automaten, regulären Ausdruck bzw. ein Eingabewort an das Applet zu schicken. Tabelle 6.1 präzisiert die nachstehenden Erläuterungen zu vier unterschiedlichen Parameterklassen.

Allgemeine Parameter Diese Parameter bestimmen allgemeine Aspekte, die für den Einbau des Applets in eine HTML-Seite wichtig sind. Dies sind beispielsweise die Hintergrundfarbe des Applets oder die Verwendung eines Vorausschauenden Layouts (siehe Abschnitt 5.4.3) für das Zeichnen der Übergangsdiagramme.

Anfangsparameter Parameter dieser Gruppe beeinflussen das Appletverhalten zu Beginn der Visualisierung. Sie legen z. B. fest, ob der Benutzer zuerst auf einen Knopf für den Animationsstart klicken muß, oder ob es Felder für die Eingabe benutzerdefinierter regulärer Ausdrücke und Eingabeworte gibt.

Algorithmenselektion Die Parameter dieser Gruppe bestimmen, welche Algorithmen visualisiert werden. Unabhängig davon, welche Visualisierung angeschaltet ist, wird jeder Generierungsalgorithmus durchgeführt und die Ausgabe von einem Algorithmus als Eingabe für den nächsten Algorithmus verwendet. Weiterhin besteht die Möglichkeit, die endlichen Automaten, die als Ergebnis der Algorithmen entstehen, auf einem Eingabewort zu simulieren.

Automatenspezifikation Statt der Bereitstellung eines regulären Ausdrucks ist es auch möglich, einen NEA oder DEA über die Appletparameter zu spezifizieren. Das Applet nutzt den Parameter `StateNr` um zu bestimmen, ob ein endlicher Automat spezifiziert ist. Dieser dient als Eingabe für den ersten Algorithmus, dessen

Allgemeine Parameter			
Parameter	**Beschreibung**	**Wertemenge**	**Default**
`bgcolor`	Hintergrundfarbe des Applets	`#`<R><G><B>	`#c0c0c0`
`Language`	Ausgabesprache	{`G`, `E`}	`G`
`Foresight`	Layoutverfahren	{`on`, `off`}	`off`

Anfangsparameter			
Parameter	**Beschreibung**	**Wertemenge**	**Default**
`StartButton`	Startknopf einblenden	{`on`, `off`}	`off`
`RegEdit`	Eingabefeld für einen RA	{`on`, `off`}	`off`
`WordEdit`	Eingabefeld für ein Eingabewort	{`on`, `off`}	`off`
`RegExp`	RA, mit dem das Applet startet	<RA>	-
`Word`	Eingabewort für eine Simulation	Σ^*	-

Algorithmenselektion			
Parameter	**Algorithmus**	**Wertemenge**	**Default**
`StepNFAVis`	RA $\rightarrow$ NEA	{`on`, `off`}	`off`
`StepCheck1`	Simulation des NEA	{`on`, `off`}	`off`
`StepEEpvis`	Entfernung der ε-Kanten des NEA	{`on`, `off`}	`off`
`StepCheck2`	Simulation des NEA ohne ε-Kanten	{`on`, `off`}	`off`
`StepDFAVis`	NEA $\rightarrow$ DEA	{`on`, `off`}	`off`
`StepCheck3`	Simulation des DEA	{`on`, `off`}	`off`
`StepMinVis`	DEA $\rightarrow$ minDEA	{`on`, `off`}	`off`
`StepCheck4`	Simulation des minDEA	{`on`, `off`}	`off`

Automatenspezifikation			
Parameter	**Beschreibung**	**Wertemenge**	**Default**
`StateNr`	Anzahl der Zustände ($\#S$)	$\mathbb{N}$	-
`InitialState`	Index des Startzustands	$[0 \dots \#S]$	`0`
`FinalStates`	Menge der Endzustände	$([0 \dots \#S])^m$	-
`From`<i>	Übergänge vom Zustand mit Index i zu weiteren Zuständen	$(\Sigma \times [0 \dots \#S])^{n_i}$	-

Tabelle 6.1: Parameterübersicht.

Animation eingeschaltet ist. Alle nachfolgenden Animationen werden weiterhin die Ausgaben der jeweils vorangegangenen Algorithmen verwenden. Sollte bereits eine Simulation eingeschaltet sein, so wird der übergebene Automat bei dieser Simulation eingesetzt.

Unterausdrücke von regulären Ausdrücken, die dem Applet als Eingabe dienen, müssen immer in Klammern „(“ und „)“ eingeschlossen sein. „|“ wird als Oder-Operator verwendet. Das Zeichen „*“ kann hinter einer schließenden Klammer als Kleene-Stern-Operator genutzt werden. Die Konkatenation zweier regulärer Ausdrücke wird durch einfaches Hintereinanderschreiben repräsentiert. Als leeres Wort ε wird ersatzweise das Zeichen „@“ verwendet. Alle anderen Zeichen können als Elemente des Eingabealphabets Σ genutzt werden. Zum Beispiel sind `(IaI|IbI)*(ccc|@)` und `(((a)|b)|c)` reguläre Ausdrücke, die vom Applet gelesen werden können.

Die Eingabe von regulären Ausdrücken zur Generierung verschiedener Automaten ist jedoch verbesserungswürdig, da diese vollständig geklammert sein müssen und die Schreibweise nicht den gängigen Konventionen entspricht. Es hat sich in der Evaluation des GaniFA-Systems (siehe Kapitel 8) gezeigt, daß Lernende dadurch schnell demotiviert werden.

6.3.2.3 Implementierungsaspekte

Das GaniFA-Applet wurde unter Verwendung des Ganimal-Frameworks implementiert. Dazu wurden die Algorithmen der Automatentheorie um Interesting Events erweitert, sowie zu jedem Algorithmus entsprechende Sichten gestaltet und entwickelt. Tiefergehende Informationen zur Implementierung und Adaptabilität der GaniFA-Anwendung können in der Diplomarbeit [Wel01] nachgelesen werden.

Weiterhin wurde im Rahmen des Ganimal-Projekts ein neuer Graphlayoutalgorithmus entworfen, der konsistente Knotenpositionen in einer animierten Sequenz von sich verändernden Graphen garantiert. In unserem Fall sind die Graphen die Übergangsdiagramme endlicher Automaten. Dieses Verfahren, das sogenannte Vorausschauende Graphlayout, wurde bereits in Abschnitt 5.4.3 näher diskutiert und erstmals im GaniFA-Applet eingesetzt.

7 Lerntheoretische Aspekte

In diesem Kapitel wird ein kurzer Überblick über grundlegende didaktische Theorien und Ansätze gegeben, welche u. a. mit multimedialen Lehr- und Lernsystemen in Zusammenhang stehen. Diese Theorien gründen vorwiegend in der Psychologie sowie in der von der Informatik beeinflußten Kognitionsforschung. Zunächst werden wir in Abschnitt 7.2 zwei lerntheoretische Konzepte, das instruktionale und das konstruktivistische Lernmodell, diskutieren. Anschließend beleuchten wir in Abschnitt 7.3 die Begriffe Multimedia und Hypermedia im Lernkontext und zeigen Problemfelder auf, die bei deren Umsetzung in Lernsystemen oftmals entstehen. Beide Abschnitte, 7.2 und 7.3, sind im wesentlichen eine Zusammenfassung ausgewählter Kapitel aus der Dissertation von Blumstengel [Blu98], dem Sammelband von Issing und Klimsa [IK97], dem Buch von Schulmeister [Sch96] sowie dem Forschungsbericht von Gerstenmaier und Mandl [GM94]. Am Ende des Kapitels stellen wir in Abschnitt 7.4 ein vierstufiges Modell explorativen Lernens vor und analysieren unsere eigenen Lernsysteme aus der Sicht dieses Modells und der diskutierten Theorien.

7.1 Terminologie

Bevor wir in die Lerntheorie einsteigen, erläutern wir einige wichtige Begriffe, die in diesem Kapitel auftreten. Im Bereich der Didaktik und Psychologie gibt es oftmals keine eindeutigen und allgemeingültigen Definitionen, wie sie der Leser aus der Informatik oder anderen naturwissenschaftlichen Gebieten gewohnt ist. Begriffserläuterungen fallen in der Literatur oft unterschiedlich aus und sind in diesem Umfeld naturgemäß sozialwissenschaftlich orientiert. Beispielsweise ist eine Definition des Begriffs *Lernen* kaum möglich, da es verschiedene Standpunkte darüber gibt, ob Lernen mit Wissens*erwerb* in Verbindung steht oder als Prozeß einer Wissens*konstruktion* verstanden wird, wie in Abschnitt 7.2 ausgeführt wird. Daher können die nachfolgenden Begriffsdefinitionen, die dem Sammelband [IK97] entnommen sind, dem Leser nur als Anhaltspunkt dienen.

Bildungssoftware „Ganz allgemein alle Arten von Software, die bildend wirken können. Der Begriff umfaßt sowohl Informationssoftware (z. B. elektronisches Lexikon), allgemeinbildende Software (z. B. Mozart und seine Werke), Lernsoftware (z. B. Sprachlernprogramme) als auch Programmanwendungen, die erst in einer bestimmten Situation bildend wirken können (z. B. Tabellenkalkulationsprogramme).“ [IK97, 481].

Lernsoftware „Software, die speziell für Lehr- und Lernzwecke konzipiert und programmiert wurde. Die didaktische Komponente liegt vor allem im Produkt d. h. in der

Software selbst und zeigt sich im Programmdesign, in der Gestaltung der Benutzeroberfläche, den vorgesehenen Feedback-Mechanismen und Interaktionsmöglichkeiten der BenutzerInnen.“ [IK97, 486].

CBT (Computer Based Training) „Eine Bezeichnung für computerunterstützte Lernprogramme, die didaktisch aus den Ansätzen des Instruktionsdesigns der 70er Jahre [siehe Abschnitt 7.2.1 – d. V.] abgeleitet werden. CBT ist für bestimmte Lernformen (z. B. Drill & Practice) gut einsetzbar.“ [IK97, 481 f.].

Hypertext Allgemein betrachtet „ein Begriff zur Beschreibung der nicht-linearen Repräsentation von Texten. [...] Im Zusammenhang mit der computerbasierten Informationsrepräsentation versteht man hierunter zum einen die Technik der nichtsequentiellen Repräsentation von Informationen in einem Netzwerk von Informationsknoten und elektronischen Verknüpfungen, zum anderen die Hypertext-Datenbasis (auch Hypertextbasis), in der die Informationsrepräsentation erfolgt.“ [IK97, 483].

Multimedia „Unter Multimedia versteht man die gleichzeitige oder zeitlich versetzte Verwendung mehrerer statischer (z. B. Text, Graphik, Tabellen) und dynamischer (z. B. Animation, Video, Audio) Medien auf einer Präsentationsplattform.“ [GH97, 418]. Abschnitt 7.3 beschreibt weitere Eigenschaften von Multimedia und deren Konsequenzen, wenn Multimedia in Lernsystemen eingesetzt wird.

Hypermedia Eine „Wortschöpfung, gebildet aus ‘Hyper’ von Hypertext und ‘media’ von Multimedia. Hypermedia ist also die Kombination von Hypertext-Funktionen und multimedialen Präsentationsformen.“ [GH97, 418].

7.2 Lerntheorien

Es existiert eine Vielzahl von didaktischen Modellen und Theorien, die sich je nach Zielsetzung, Inhalten und Lehr- bzw. Lernbedingungen unterscheiden, angefangen vom passiven, rezeptiven Lernen mit permanentem Feedback bis hin zum selbständigen, explorativen Lernen. Die beiden Standpunkte stellen Extrempositionen in bezug auf die lerntheoretische Fundierung dar. Im ersten Fall stehen die *Vermittlung* von Wissen, die Instruktion, das Lehren und eine deduktive Vorgehensweise im Vordergrund (Instruktionsparadigma). Im zweiten Fall wird eine induktive Vorgehensweise sowie die Anleitung, Hilfestellung und Unterstützung zu selbständigem Lernen favorisiert (Problemlösungsparadigma). Hier wird auf die *Erarbeitung* von Wissen durch den Lernenden Wert gelegt (vgl. [Iss97, 197 ff.]). In den nachfolgenden Abschnitten werden zwei diesen Ausprägungen entsprechende Theorien diskutiert und einander gegenübergestellt: der auf dem Behaviourismus basierende Instruktionalismus und der auf der Kognitionstheorie gründende Konstruktivismus.

7.2.1 Instruktionalismus

Das instruktionalistische Lernmodell basiert hauptsächlich auf Thesen der behaviouristischen Psychologie (vgl. [Sch96, 107]). Diese ging in ihrer ursprünglichen Form davon aus, daß auf bestimmte Reize (**S**timuli) ebenso bestimmte Verhaltensreaktionen (**R**esponse) folgen, und daß sich solche S-R-Verbindungen aneinanderreihen und als sogenannte *gepaarte Assoziationen* verinnerlicht werden. Weiterführende Modelle ließen spontan auftretende Verhaltensreaktionen zu, deren Auftrittswahrscheinlichkeit durch sogenannte Verstärker, z. B. Erfolgserlebnisse oder Lob, erhöht werden konnte. Der Prozeß des Verinnerlichens folgte dem Paradigma der operanten Konditionierung und wurde durch die gezielte Belohnung im Falle einer erwünschten Reaktion gesteuert. Unerwünschte Reaktionen blieben demnach unbelohnt. Im Zuge dieser Annahmen wurde der Lernstoff in kleinste Einheiten zerstückelt. Nach jeder Einheit hatte der Lernende eine Antwort zu geben, die mit einer vorgegebenen Antwort verglichen wurde. Richtige Antworten führten zu einer Verstärkung. Falsche Antworten hatten eine Wiederholung der atomaren Lerneinheit zur Folge und mußten möglichst vermieden werden. Dies führte dazu, daß die Lerneinheiten meist mit Lernhilfen durchsetzt und sehr suggestiv waren (vgl. [Sch96, 87]).

Bezüglich der Motivation des Lernenden spricht man hier von *extrinsischer* Motivation. Der Grund für das Lernen liegt außerhalb des eigentlichen Lernbereichs, d. h. in diesem Fall liegt der Anreiz in der Belohnung des Lernenden im Falle einer korrekten Antwort. Bei der *intrinsischen* Motivation besteht im Gegensatz dazu ein interner Anreiz (z. B. Neugier). Der den Anreiz auslösende Faktor wird subjektiv als notwendig oder ausreichend interessant betrachtet, um einen bestimmten Sachverhalt nachvollziehen zu wollen (vgl. [Blu98, 141]).

7.2.1.1 Programmierte Instruktion

Eine Realisierung dieses behaviouristischen Konzepts mit technischen Medien ist die von Skinner entwickelte *Programmierte Instruktion* [Ski58], die in den 60er und 70er Jahren weite Verbreitung fand. Der Unterricht sollte durch die Programmierte Instruktion nicht nur wiederholbar gemacht werden. Aufgrund der Objektivierung dieser Lehrmethode sollte eine Bewertung des Lernerfolgs auch gerechter sein (vgl. [Kli97, 16]).

Wissen wird hier als eine Menge von Fakten (deklaratives Wissen) und Regeln (prozedurales Wissen) verstanden, die unabhängig von einzelnen Personen objektiv existiert. Es wird als externe Information aufgefaßt, die durch den Lehrenden transportiert wird. Demzufolge steht die Vermittlung bzw. der Erwerb von Wissen im Mittelpunkt instruktionalistischen Denkens. Der Lernende wird hier fast vollständig ignoriert. Seine Aufgabe ist auf die Rezeption des Lernstoffes beschränkt. Dabei werden Problemlösungsfähigkeiten kaum vermittelt (vgl. [MGR97, 167 f.], [Blu98, 110 f.]).

Aus diesem Ansatz sind Autorensysteme (Drill & Practice) hervorgegangen, die auch heute noch häufig zur Entwicklung von multimedialer Lehr- und Lernsoftware eingesetzt werden [KWD00].

7.2.1.2 Instruktionstheorie

Aufgrund der Kritik am assoziativen Lernen des Behaviourismus entstand in den 70er und 80er Jahren die *Instruktionstheorie* (engl. Instructional Design, ID) [GBW79]. Das Ziel dieser Theorie lag in der Vermeidung der extremen Atomisierung der Lerneinheiten und in der variableren Gestaltung von Lehrmethoden, um eine möglicherweise „effektivere Form der Individualisierung zu erreichen“ [Sch96, 105]. Ferner wollte man „durch Deduktion aus Lernzielen und Regeln zu einer Automatisierung der Produktion von Lerneinheiten kommen“ [Sch96, 105], d. h. die Instruktionstheorie ist auch als Entwicklungsmodell zu verstehen. Die grundlegenden Ansätze der instruktionalistischen Lerntheorie blieben allerdings in den wichtigsten Punkten mit denen des Behaviourismus gleich. Für das Verstehen und die Interpretation des Lehrstoffes bietet auch die Instruktionstheorie keine angemessene Unterstützung. Die hauptsächlichen Kritikpunkte an der Instruktionstheorie sind ihr Objektivismus und Reduktionismus. „Unter Reduktionismus wird dabei die starke Orientierung auf Lernziele und Sublernziele im Rahmen der Task Analysis [Erarbeitung von Lernzielen durch Inhalts- und Aufgabenanalysen – d. V.] verstanden, welche die Komplexität von Wissensbereichen und Querverbindungen zwischen den Einzelinformationen vernachlässigt.“ [Blu98, 152]. Wohl aus diesem Grunde haben sich ID-Systeme in der Praxis nicht durchsetzen können. Heute gilt der behaviouristische Lernansatz, die Programmierte Instruktion eingeschlossen, als veraltet. Es gibt allerdings einige Lerngebiete, in denen die behaviouristische Sichtweise eine durchaus angemessene und sinnvolle Anwendung findet. Dies sind beispielsweise Aufgaben, in denen das Erinnern von Fakten im Vordergrund steht, wie etwa Vokabeltraining (vgl. [Blu98, 111], [Sch96, 88]).

7.2.2 Konstruktivismus

Der Konstruktivismus begründet sich vorwiegend auf Theorien der kognitiven Psychologie. Der bekannteste Vertreter der Kognitionstheorie ist Jean Piaget. Er beschreibt Lernprozesse als Austauschvorgänge mit der Umwelt. Diese werden von ihm als *Akkomodation* und *Assimilation* bezeichnet, „als die Anpassung der erworbenen kognitiven Konzepte an neue pragmatische Gegebenheiten und als die Hereinnahme externer Objekte und Zustände in die inneren Strukturen des Individuums unter Modifikation der vorhandenen kognitiven Strukturen.“ [Sch96, 65]. Es wird bereits bei dieser Aussage ersichtlich, daß der Lernende im Gegensatz zum Behaviourismus als eigenständiges Individuum angesehen wird, welches externe Reize aktiv und selbständig verarbeitet. Als Folge dessen wird Lernen in der Kognitionstheorie als Informationsverarbeitungsprozeß verstanden. Aber auch hier wird von einem externen und objektiv existierenden Wissen ausgegangen. Somit haben die Theorien der kognitiven Psychologie ebenfalls objektivistische Tendenzen (vgl. [Blu98, 112]).

Der Konstruktivismus lehnt „jedoch die Annahme einer Wechselwirkung zwischen der externen Präsentation [des Wissens – d. V.] und dem internen Verarbeitungsprozeß [des Individuums – d. V.] ab“ [Blu98, 115] und bildet so eine Gegenposition zum Objektivismus, wie in Abschnitt 7.2.2.2 näher beschrieben wird. Zunächst möchten wir zwei

wichtige Lernkonzepte beleuchten, die aus der erkenntnistheoretischen Theorie der Kognition entstanden sind: Mikrowelten und entdeckendes Lernen.

7.2.2.1 Mikrowelten und entdeckendes Lernen

Das Prinzip des *entdeckenden* oder *explorativen Lernens* wurde von Jerome S. Bruner [Bru61] zu Beginn der 60er Jahre begründet. Im Vordergrund stehen „der an der Heuristik menschlichen Denkens orientierte Erkenntnisprozeß, der konzeptgeleitete Denkprozeß und das konstruktive Problemlösen“ [Sch96, 65]. Die Theorie des entdeckenden Lernens unterstreicht nach Blumstengel die folgenden Lernaspekte (vgl. auch [Ede96]):

1. *Entdeckendes Lernen wird durch den Lernenden selbst gesteuert.*

2. *Statt alle relevanten Informationen fertig strukturiert zu präsentieren, muß der Lernende Informationen finden, priorisieren und neu ordnen, bevor er daraus Regeln ableiten und Probleme lösen kann.*

3. *Die Exploration wird geleitet von Neugier und Interesse des Lernenden. Er soll Lösungen für interessante Fragen entwickeln, statt Fakten auswendig zu lernen. Besonders wichtig ist dabei, wie bei jeder Form des selbstgesteuerten Lernens, ein hoher Grad an intrinsischer Motivation. Der Stellenwert des impliziten Lernens* [unbewußtes Lernen – d. V.] *und der Intuition wird ebenfalls betont. Dem Entdeckenden Lernen wird insgesamt eine motivierende Wirkung zugesprochen.* [...]

4. *Ziel des Lernens ist die Ausbildung der Problemlösungsfähigkeit.* [Blu98, 112 f.]

Mikrowelten beschreiben in sich geschlossene, künstliche Umgebungen mit eigenen Regeln, die interaktiv veränderbar sind. Lernende können durch verschiedene Eingriffe auf die Umgebung einwirken oder bestimmte Bestandteile manipulieren (vgl. [Sch96, 46 f.]). Mikrowelten sind somit den bekannten Simulationen, die beispielsweise in Lernsystemen für die Bereiche der Physik oder Elektrotechnik oft Verwendung finden, sehr ähnlich. Der Unterschied ist, daß sie quasi als Simulation zweiter Ordnung die Konstruktion von Modellen propagieren, anstatt nur die Arbeit mit einem bestehenden Modell zu unterstützen. Die Methodologie des entdeckenden Lernens mit Hilfe einer Simulationsumgebung oder Mikrowelt beschreibt Blumstengel demnach wie folgt:

1. *Formulierung von Hypothesen auf Basis des vorhandenen Vorwissens sowie ggf. weiterer, durch die Lernumgebung angebotener Informationen,*

2. *Test der Hypothese(n) am Modell,*

3. *Überprüfung, inwieweit die Ergebnisse des Tests mit den Hypothesen vereinbar sind sowie*

4. *gegebenenfalls Modifikation der Hypothesen oder des Testdesigns.* [Blu98, 44]

Blumstengel führt weiter aus, daß Mikrowelten und Simulationen daher das aktive und explorative Lernen fördern und bei guter Gestaltung motivierend wirken. Exploratives Lernen ist gut mit der konstruktivistischen Sichtweise zur Gestaltung von Lernumgebungen zu vereinbaren, die wir im folgenden ausführlicher diskutieren.

7.2.2.2 Konstruktive Wissensaneignung

Im Mittelpunkt der konstruktivistischen Position steht die Auffassung, daß Wissen im Erkennensprozeß konstruiert wird. Es existiert in Relation zum Lernenden nicht autonom, sondern wird von diesem dynamisch generiert und kann deswegen nicht auf traditionelle Weise vermittelt werden. Das Wissen muß vom Lernenden selbständig in dessen bestehende mentale Struktur und Wissenskonstrukte integriert werden. Wissensübermittlung ohne eigene Rekonstruktion sowie Objektivität und subjektunabhängiges Denken sind aus dieser Sichtweise nicht möglich. Galt die Wissensvermittlung noch als das Kerngebiet des Instruktionalismus, legen Konstruktivisten ihr Hauptaugenmerk auf höhere Lern- und Denkprozesse, wie etwa Interpretieren und Verstehen (vgl. [Sch96, 153]).

Offensichtlich entspricht diese Sichtweise eher einem Programm als einer kohärenten Theorie, geschweige denn einem Modell zur Entwicklung von Lernumgebungen. In der Tat wird sie in der Literatur auch als radikaler Konstruktivismus bezeichnet und erfuhr teilweise heftige Kritik; man warf ihr u. a. Fundamentalismus vor. Vom Standpunkt der Instruktion aus halten radikale Konstruktivisten lediglich selbstgesteuertes, kollektives Lernen für richtig. Die gemäßigtere Ausprägung des Konstruktivismus, auch als pragmatischer, moderater Konstruktivismus bezeichnet, sieht die Aufgabe einer Lehrperson hingegen in der Unterstützung bzw. Anregung des individuellen Konstruktionsprozesses (nach [GM94, 45]).

Dieser Prozeß kann und darf von der Lehrperson allerdings in keinster Weise manipuliert werden. Folglich wird hier Wert auf die Schaffung und Gestaltung von stimulierenden *Lernumgebungen* gelegt, welche Lernenden die Möglichkeit bieten, individuelle Wissenskonstrukte zu generieren. Im verbleibenden Teil dieses Kapitels gehen wir von einem moderaten, konstruktivistischen Ansatz aus, wenn von Konstruktivismus die Rede ist. Zusammenfassend lassen sich die wichtigsten konstruktivistischen Grundannahmen über den Wissenserwerb wie folgt beschreiben (nach [GM94, 29], [Blu98, 118], [JMM93]):

- Lernen ist aktive Wissenskonstruktion und vollzieht sich in Abhängigkeit von Vorwissen sowie aktuellen mentalen Strukturen und Überzeugungen.
- Wissen stammt nicht aus einer externen Quelle, sondern wird vom Lernenden intern erzeugt.
- „Wissen an sich ist durch den Lehrer nicht vermittelbar.“ [Blu98, 118]. Vielmehr fördert er die aktive Wissenskonstruktion des Lernenden. Wichtig ist das Aushandeln von Bedeutungen, das auch durch kollaborative Prozesse zwischen Lehrenden und Lernenden geschehen kann.
- Fehlt der Bezug zu einem relevanten Kontext, dann ist das Wissen bzw. die Information für den Lernenden wenig bedeutend (s. u.).

- Zur Überprüfung der eigenen Lernhandlung werden metakognitive Fähigkeiten als wichtig angesehen, d. h. das Vermögen, den eigenen Prozeß der Wissenserarbeitung und -konstruktion zu verstehen und zu regulieren. Der Lernende selbst ist seine eigene Kontrolle.

7.2.2.3 Situiertes Lernen

Die Situation, in der die Wissenskonstruktion stattfindet, spielt in diesem Zusammenhang eine große Rolle. Das Konzept des *Situierten Lernens* ist in der Literatur nicht einheitlich definiert [MGR97, 168]. Im allgemeinen wird Lernen immer als situiert verstanden, als ein Vorgang, in dem interne Faktoren mit externen, situativen Komponenten interagieren. Situiertes Lernen wird meist soweit gefaßt, daß auch Wechselbeziehungen zwischen Menschen und ihr jeweiliger kultureller und sozialer Kontext bedeutsame Aspekte der Situation darstellen. Nach Mandl et al. [MGR97, 168] kritisieren Vertreter dieses Konzepts vor allem den traditionellen Frontalunterricht an Schulen und Universitäten, der ihrer Ansicht nach wenig mit der Findung realer Problemlösungsstrategien zu tun hat und die Lernenden isoliert arbeiten läßt. Ihre zentrale Forderung liegt vielmehr darin, Lern- und Anwendungssituationen möglichst gleichartig zu kreieren. Nur so sei mit effektivem Wissenstransfer zu rechnen.

Nach Gerstenmaier und Mandl [GM94, 44] sind konstruktivistische Instruktionsansätze eng mit den sogenannten neuen Technologien und Medien verbunden. Manche Autoren sprechen von diesen sogar als „Werkzeuge für den Konstruktivismus“ [KC93]. Voraussetzung sei hierbei, daß sie den Lernenden Spielraum für eigene Konstruktionen bieten, und daß die Lernenden diesen Spielraum auch akzeptieren und nutzen. „Die konstruktivistische Auffassung vom aktiven Lerner und situierter Kognition wird durch diese neuen Technologien im Unterricht effektiv unterstützt, sie bietet vor allem bei Formen des selbstgesteuerten Lernens in Schule, Hochschule und Weiterbildung wichtige Perspektiven.“ [GM94, 44].

7.2.3 Schlußfolgerungen

In der Literatur gab und gibt es zwischen den Vertretern der diskutierten Lerntheorien heftige Debatten und wechselseitige Kritik. Schulmeister diskutiert diese Auseinandersetzungen ausführlich [Sch96, 150 ff.]. Wir greifen einen Aspekt heraus, der im Kontext dieser Arbeit besonders relevant ist: Wie weit und auf welche Art und Weise soll eine Lehrperson oder ein Lernsystem den Prozeß des Lernens steuern und unterstützen? Im folgenden versuchen wir, eine Antwort auf diese Fragestellung zu finden.

7.2.3.1 Lernerkontrolle vs. Programmkontrolle

Unter Lernerkontrolle wird im Bereich des computergestützten Lernens „die Kontrolle des Lernenden über die Auswahl und über die Sequenzierung der Inhalte bzw. Übungen verstanden“ [Sch96, 139]. Nach Schulmeister argumentieren viele Autoren, daß Lernsoftware, in welcher der Computer als Korrektor und Autorität auftritt, entmutigend wirkt

und wenig erfolgversprechend ist. Weiterhin vermuten sie einen positiven Effekt der Lernerkontrolle auf das Erreichen kognitiver Lernziele. Allerdings gibt es Argumentationen für eine graduelle Einschränkung der Lernerkontrolle in Abhängigkeit von Vorwissen, Alter, Lernfortschritt des Lernenden usw. Eine reine Lenkung des Lernenden durch den Computer (Programmkontrolle) wird meist abgelehnt. Viele Studien befassen sich mit diesem Problemfeld, kommen aber wegen der extremen Abhängigkeit der Lerner- bzw. Programmkontrolle von den jeweiligen Lernumgebungen sowie deren Unvergleichbarkeit zu widersprüchlichen Ergebnissen.

7.2.3.2 Instruktionsmethoden

Wie wir bereits gesehen haben, schließt eine gemäßigt konstruktivistische Auffassung vom Lernen die Anleitung und Unterstützung des Lernenden durch Lehrende bzw. Lernsysteme keineswegs aus. Wichtig sind hierbei die Ziele der Instruktion: Vom instruktionalistischen Standpunkt aus gesehen liegen diese in der Festlegung der Lernziele, in der Wissensvermittlung und der Lernkontrolle; vom moderaten konstruktivistischen Blickwinkel aus gesehen sind diese „die Aktivierung des Lernenden, die Anregung des (natürlichen und individuellen) Lernprozesses sowie die Metakognition und Toleranz für andere Perspektiven“ [Blu98, 116]. Der Vollständigkeit halber führen wir nun eine Charakterisierung verschiedener Typen komplexer Instruktionsstrategien von Weinert [Wei96] auf, wobei deren Reihenfolge grob den in den letzten Jahren beobachtbaren Wechsel vom Instruktionsparadigma zum Paradigma offener Lernumwelten widerspiegelt:

Direkte Instruktion Unter diesem Begriff versteht man eine weitgehend externe Steuerung des Lernens, etwa durch eine Lehrperson oder ein Lernsystem. Weinert betont, daß es sich bei dieser Strategie nicht um bornierten Paukunterricht handelt, sondern um eine didaktisch anspruchvolle Instruktionsform, in welcher der Sachverstand der Lehrperson dazu verwendet wird, die Lern- und Leistungsfortschritte der Lernenden zu maximieren.

Adaptive Instruktion Dieser Typus bezeichnet den Versuch, die didaktischen Hilfen so an die kognitiven, motivationalen und affektiven Unterschiede der Lernenden anzupassen, daß der Einzelne möglichst optimal davon profitiert und bestmöglich gefördert wird (vgl. [Wei96]). Im Fall von Lernsystemen bezieht sich Adaptivität auf die Frage, inwieweit das System in der Lage ist, den Unterstützungsbedarf des Lernenden zu erkennen und diese Erkenntnis in eine geeignete Lehrtätigkeit zu transformieren (vgl. [Leu97, 141]).

Tutoriell unterstütztes Lernen Eine Spezialform der adaptiven Instruktion ist das tutorengestützte Lernen. Im Vordergrund stehen die sogenannten Intelligenten Tutoriellen Systeme (ITS). Diese versuchen, das Verhalten einer Lehrperson in Abhängigkeit eines Wissensgebiets, eines Modells vom Lernenden, mehrerer didaktischer Strategien und einer Kommunikationskomponente nachzubilden. Hier vereinigen sich Ansätze der Kognitionstheorie und der Künstlichen Intelligenz (vgl. [Sch96, 167]).

Kooperatives Lernen Das Konzept des oben diskutierten situierten und kontextuierten Lernens führt direkt zu den sog. kooperativen Lernformen: „Bei der gemeinsamen Bearbeitung eines Problems müssen Lernende sich mit den Auffassungen der Kooperationspartner auseinandersetzen und können auf diese Weise zu einem tieferen, die unterschiedlichen Sichtweisen der Partner einbeziehenden Verständnis eines Sachverhalts gelangen.“ [Ter97, 131].

Selbständiges Lernen Prinzipiell drückt der Begriff *Selbständiges Lernen* aus, daß Lernende ganz im Gegensatz zum traditionellen Unterricht ihre eigenen Lehrer sein sollen. Paris und Newman [PN90] argumentieren, daß „in traditional instruction, the teacher is predominantly active and the students are passive. This imbalance should be reversed. Self-generated, self-organized, self-controlled and self-evaluated learning (in contrast to learning that is directed by others and controlled by the teacher) is perceived as an important, if not the essential, prerequisite for understanding, insight and discovery“. Andere Autoren (vgl. [WH95]) erkennen im selbstgesteuerten Lernprozeß jedoch Defizite

- in der systematischen Konstruktion von Wissen,
- im Abstraktionsniveau der gelernten Information,
- in den Lernstrategien und
- in der Korrektheit der gelernten Information.

Einigkeit herrscht in der Literatur dahingehend, daß selbständiges und selbstkontrolliertes Lernen in der Regel eine intensive und wohlüberlegte didaktische Einführung benötigt.

Eine Antwort auf die Frage, was nun das bessere Instruktionsmodell sei, wird von Weinert und Helmke [WH95] gegeben: „An old piece of educational wisdom is that no single method of instruction is the best for all students and for all learning goals, and that even very effective instructional procedures can have deficits with respect of single criteria.“ Meist werden Mischformen der Instruktion und sogar der zugrundeliegenden Lerntheorien propagiert, allerdings mit zunehmender Tendenz zum selbständigen Lernen und zu konstruktivistischen Ansätzen. Demzufolge wird für Anfänger ein eher extern gesteuertes Lernen mit geringerer Lernerkontrolle und mit instruktionalen Komponenten empfohlen, wohingegen Fortgeschrittenen Raum für selbständiges und exploratives Lernen in konstruktivistisch orientierten Lernumgebungen gegeben werden sollte.

Eine Zusammenfassung der in diesem Abschnitt diskutierten Konzepte zeigt Tabelle 7.1. Neben Eigenschaften des von der Instruktionstheorie favorisierten Behaviourismus und des Konstruktivismus werden zusätzlich einige prägnante Aspekte des Kognitivismus aufgezählt.

Kategorie	**Behaviourismus**	**Kognitivismus**	**Konstruktivismus**
Hirn ist ein	*Passiver Behälter*	*Informationsverarbeitendes 'Gerät'*	*Informationell geschlossenes System*
Wissen wird	*Abgelagert*	*Verarbeitet*	*Konstruiert*
Wissen ist	*Eine korrekte Input-Output-Relation*	*Ein adäquater interner Verarbeitungsprozeß*	*Mit einer Situtation operieren zu können*
Lernziele	*Richtige Antworten*	*Richtige Methoden zur Antwortfindung*	*Komplexe Situationen bewältigen*
Paradigma	*Stimulus-Response*	*Problemlösung*	*Konstruktion*
Strategie	*Lehren*	*Beobachten und Helfen*	*Kooperieren*
Lehrer ist	*Autorität*	*Tutor*	*Coach, (Spieler)Trainer*
Feedback	*Extern vorgegeben*	*Extern modelliert*	*Intern modelliert*
Interaktion	*Starr vorgegeben*	*Dynamisch in Abhängigkeit des externen Lehrmodells*	*Selbstreferentiell, zirkulär, strukturdeterminiert (autonom)*
Programmmerkmale	*Starrer Ablauf, quantitative Zeit- und Antwortstatistik*	*Dynamisch gesteuerter Ablauf, vorgegebene Problemstellung, Antwortanalyse*	*Dynamisch, komplex vernetzte Systeme, keine vorgegebene Problemstellung*
Softwareparadigma	*Lernmaschine*	*Künstliche Intelligenz*	*Sozio-technische Umgebungen*
'idealer' Softwaretypus	*Tutorielle Systeme, Drill & Practice*	*Adaptive Systeme, ITS*	*Simulationen, Mikrowelten, Hypermedia*

Tabelle 7.1: Lernparadigmen und Softwaretypologie ([BP94], zit. und erw. von [Blu98, 108]).

7.3 Lernen mit Multi- und Hypermedia

Dieser Abschnitt beschäftigt sich mit der Darstellung der Begriffe *Multimedia* bzw. *Hypermedia* und zeigt abschließend die bedeutendsten Problemfelder auf, die sich aus dem Kontext des Lehrens und Lernens mit Multimedia ergeben. Das in diesem Zusammenhang zu nennende Hypertextkonzept wird hier nicht weiter erläutert, eine Diskussion darüber kann beispielsweise bei Tergan [Ter97] nachgeschlagen werden. Wir beschränken uns auf die Definition in Abschnitt 7.1.

7.3.1 Multimedia

In Abschnitt 7.1 zur Terminologie wurde eine der klassischen und weitverbreitetsten Definitionen von Multimedia gegeben. Es existieren viele verschiedene Definitionsansätze, die oftmals darauf abzielen, bestimmte „Multimedia"-Applikationen auszugrenzen, welche auf technischen Vehikeln, wie etwa den Videodisks der 80er Jahre, basieren (vgl. [Blu98, 70]). Neuere Definitionen des Begriffs *Multimedia* haben folgende Punkte gemeinsam:

- Speicherung in einer digitalen Form.
- Verwendung einer bestimmten Zahl von Medienarten (siehe Abschnitt 7.3.1.1).
- Förderung von Interaktivität (siehe Abschnitt 7.3.1.2).
- Computerbasierte Integration der Informationen.

In den letzten Jahren haben medienwissenschaftliche Beiträge die Ungenauigkeit der zum Teil „theorielosen“ [Wei97b, 65] Definitionen von Multimedia kritisiert und eine differenziertere Beschreibung multimedialer Angebote versucht. Weidenmann [Wei97b, 67] schlägt alternativ zum Multimediabegriff eine Charakterisierung von Lernsystemen hinsichtlich der Kategorien *Kodierung* und *Modalität* vor. Mit Kodierung wird das verwendete Symbolsystem (verbal, piktoral, Zahlensystem etc.) bezeichnet. *Multikodal* sind Angebote mit unterschiedlichen Symbolsystemen, beispielsweise ein Text mit Bildern. Die Kategorie der Modalität beschreibt das angesprochene Sinnesorgan (visuell, auditiv, haptisch, olfaktorisch oder gustatorisch). Ein *multimodales* Angebot spricht mehrere Sinnesmodalitäten der Nutzer bzw. Lernenden an, z. B. ein Video mit Ton. Unter *Medium* versteht man ein Objekt, technisches Gerät oder eine Konfiguration, mit der sich Botschaften speichern und präsentieren lassen.

7.3.1.1 Medien

Die unscharfe Definition des Multimediabegriffs geht auf die außerordentlich vieldeutige Charakterisierung des Begriffs *Medium* zurück. Steinmetz [Ste95] klassifiziert Medien auf einer mehr technologischen Basis u. a. in:

- Perzeptionsmedien, d. h. das angesprochene Sinnesorgan betreffend (z. B. visuell, auditiv),
- Repräsentationsmedien, welche die rechnerinterne Codierung tangieren (z. B. GIF, ASCII),
- Präsentationsmedien, d. h. Ein- und Ausgabemedien (z. B. Tastatur, Bildschirm), und
- Speichermedien (z. B. Festplatte, CD-ROM).

Weitverbreitet ist auch die Unterscheidung in statische, zeitunabhängige Medien, z. B. Text oder Graphik, und dynamische, zeitabhängige Medien wie etwa Video und Audio. Im folgenden werden vorwiegend solche Medien näher diskutiert, die in unseren Systemen und im Ganimal-Framework von entscheidender Bedeutung sind. Zu diesen gehören Perzeptionsmedien wie Abbildungen, Animationen und Audio.

Abbildungen Unter Abbildungen verstehen wir statische Medien: Graphiken, Fotos, Zeichnungen und Diagramme. Für Lernsysteme stellt sich die Frage, wie und mit welchem Erfolg Abbildungen von Lernenden genutzt werden. Abbildungen erfüllen drei didaktische Aufgaben (vgl. [Wei97a, 108]):

1. *Zeigefunktion*: Sie können einen Gegenstand als Ganzes oder einen bestimmten Teil eines Gegenstands zeigen. Es gilt, die Aufmerksamkeit des Lernenden auf die wichtigsten Aspekte des Gegenstands zu lenken (Fokus), ohne den umfassenden Eindruck einzuschränken.

2. *Situierungsfunktion*: Sie können ein Szenario oder einen anderen „kognitiven Rahmen“ bereitstellen. Betrachten wir ein Sprachlernsystem als Beispiel, dann erkennen Lernende die auf einem Foto dargestellte Szene etwa als ein Verkaufsgespräch oder einen Nachbarschaftsstreit.

3. *Konstruktionsfunktion*: Sie können dem Lernenden helfen, ein mentales Modell zu einem Sachverhalt zu konstruieren, und Unvertrautes bzw. Unanschauliches verständlich machen. Abbildungen informieren den Lernenden visuell sowohl über die Elemente des Modells im einzelnen, als auch über das Zusammenspiel dieser Elemente. Aufgrund der vielfältigen Zustandsänderungen, die sich aus diesem Vorgehen ergeben, lassen sich mentale Modelle am besten durch Animationen (s. u.) visualisieren.

Der Einsatz von Abbildungen unterstützt den Prozeß des Wissenserwerbs. Nelson et al. [NRW76] konnten nachweisen, daß Abbildungen besser behalten werden als der entsprechende Begriff, der den abgebildeten Gegenstand

benennt. Man spricht vom sog. „Bildüberlegenheitseffekt“. Dies gilt zumindest für einfachere Begriffe, jedoch nicht unbedingt für abstrakte Begriffe, da oft keine adäquate, natürliche visuelle Repräsentation eines solchen Begriffs existiert. Auf diesen Ergebnissen bauen mehrere Theorien auf, z. B. die Doppelcodierungs-Theorie von Paivio [Pai86] oder die Theorie der Hemisphären-Spezialisierung von Ornstein [Orn74]. Ebenfalls empirisch sehr gut belegt ist die positive Wirkung von Illustrationen auf das Behalten von Text. Die simultane Repräsentation von Abbildung und Text führt zu einem „Mapping“ der beiden Darstellungsformen, die von Lernenden aufeinander bezogen und in bestehendes Wissen integriert werden (vgl. [Wei97b, 71]).

Animationen Viele Sachverhalte lassen sich mit statischen Abbildungen nur unzureichend darstellen. Beispiele sind etwa Bewegungsabläufe in der Sportwissenschaft, dynamische Simulationen im technisch-naturwissenschaftlichen Bereich oder die Visualisierung von Algorithmen und Programmen in der Informatik. Park und Hopkins [PH94] sprechen dem Einsatz dynamischer Animationen in Lernsystemen sechs didaktische Funktionen zu:

1. *Demonstration sequentieller Abläufe beim Lernen von Prozeduren.*

2. *Simulation kausaler Modelle komplexer Verhaltenssysteme.*

3. *Explizite Repräsentation unsichtbarer Funktionen und unsichtbaren Verhaltens.*
4. *Illustration einer Aufgabe, die verbal nur schwer zu beschreiben ist.*
5. *Visuelle Analogie für abstrakte und symbolische Konzepte.*
6. *Mittel, um die Aufmerksamkeit für bestimmte Aufgaben zu erhalten.* [Sch96, 53]

Weidenmann [Wei97b, 73] erwähnt das Risiko, daß Animationen überfrachtend wirken können und dadurch die Informationsaufnahme des Lernenden stören. Risikomindernden Effekt haben eine ausreichende Animationskontrolle, eine mentale Vorbereitung des Lernenden und synchrone Schrift- bzw. Sprachkommentierungen. Es scheint nur wenige aussagekräftige empirische Studien über die Lernwirksamkeit von Animationen zu geben. Ein Beispiel ist die Diplomarbeit [Zub99]. Sie beschreibt einen empirischen Vergleich zwischen Animationen und Standbildern und postuliert einen kleinen Vorteil der Animationen. Kapitel 8 dieser Arbeit diskutiert eine Evaluation unseres elektronischen Textbuchs über die Theorie der endlichen Automaten, welches generativ erzeugte Animationen des GaniFA-Applets beinhaltet.

Audio Die Integration der auditiven Modalität (Sprache, Musik, Ton) in multimediale Anwendungen eröffnet neue Möglichkeiten, das Lernen zu unterstützen. Aktuelle Studien sprechen nach Weidenmann dafür, daß die bereits erwähnte Überlast durch multimediale Angebote sich durch eine Verteilung der Information auf unterschiedliche Sinnesmodalitäten und Kodierungen verringern läßt (vgl. [Wei97b, 73]). Folgerichtig sollte Audio immer dann herangezogen werden, wenn die visuelle Modalität zu stark belastet ist. Weiterhin können bestimmte Signaltöne die Aktivität des Lernenden begleiten, steuern sowie auf wichtige Ereignisse hinweisen.

Weidenmann führt weiter aus, daß die Forschung in diesem Bereich noch nicht sehr weit fortgeschritten ist. So können allgemeine Aussagen über Vor- bzw. Nachteile der Anwendung von Audio noch nicht statistisch gesichert abgegeben werden. Es konnte allerdings empirisch belegt werden, daß eine audiovisuelle Präsentation von Lerninhalten im Vergleich mit einer reinen Textdarstellung von Probanden als weniger ermüdend wahrgenommen wurde, und daß diese in einem Verständnistest höhere Leistungen erbrachten als die Vergleichsgruppe ([Pyt94], zit. nach [Wei97b, 73]).

7.3.1.2 Interaktivität

Unter *Interaktivität* versteht man umfassend die Eigenschaften eines Computersystems, die dem Benutzer Eingriffs- und Steuermöglichkeiten bieten. Das Interaktionspotential stellt eines der wichtigsten Charakteristika für multi- und hypermediale Lernformen dar. Der Grad an Interaktion entscheidet im wesentlichen darüber, wie der Lernende ein System erlebt. Weiterhin gilt ein hohes Potential an Interaktivität in einem Lernsystem als *der* Erfolgsfaktor. Im Zusammenhang mit Multimedia wird mit dem Begriff der Interaktivität meist der wahlfreie Zugriff auf die Inhalte bzw. Medien in Verbindung gebracht (vgl. [Blu98, 146], [Sch96, 39 ff.]).

Vertreter des Forschungsgebiets der HCI (Human Computer Interaction) schlagen folgende Unterscheidung allgemeiner Interaktionsarten vor (vgl. [PSB+94], zit. nach [Blu98, 145]):

- Kommandozeileneingabe
- Formulare und Spreadsheets
- Menüs
- Frage-Antwort-Dialoge
- Natürlichsprachliche Dialoge
- Direkte Manipulation. Dieser Punkt hat im Kontext dieser Arbeit den höchsten Stellenwert. Eigenschaften der *direkten Manipulation* sind
  - die permanente Darstellung der relevanten Objekte,
  - die mögliche unmittelbare physische Manipulation der Objekte (z. B. mit Hilfe der Maus) und
  - die Ausführung inkrementeller sowie reversibler Operationen in Echtzeit.

  Beispielsweise lassen sich durch Anklicken eines bestimmten Teils einer Abbildung weitere Informationen in Form eines zusätzlichen Erläuterungsfensters oder einer Erklärung in gesprochener Sprache abrufen.

Vor einem eher lerntheoretischen Hintergrund klassifiziert Schulmeister [Sch96, 42 f.] den Interaktionsbegriff in drei verschiedene Formen: Wenn ein Lernender nur Antworten auf vom System dargestellte Stimuli gibt, dann spricht man von *reaktiver* Interaktion. Im Gegensatz dazu unterstreicht die sogenannte *proaktive* Interaktion die aktive Konstruktion und die generierende Tätigkeit des Lernenden. Die erste Form ist behaviouristisch geprägt und verlangt vom Lernenden, sich an das Lernsystem anzupassen; der proaktive Stil offeriert freier gestaltete Lernumgebungen im konstruktivistischen Sinn. Einen weiteren Ansatz beschreibt die *wechselseitige* Interaktion, in der eine wechselweise Anpassung zwischen Lernendem und Lernsystem stattfinden soll. Dieser kann derzeit aber noch nicht adäquat in die Praxis umgesetzt werden.

Sanktionsfreiheit Schulmeister weist zudem auf ein weiteres Merkmal der Interaktion hin, welches gemeinhin wenig Beachtung findet, nämlich daß sie normalerweise vollkommen frei von Bewertungen und sozialen Konsequenzen ist. In sozialen Wechselbeziehungen zwischen Menschen, z. B. zwischen Lehrer und Schüler, sind einmal vermittelte Eindrücke nicht ohne weiteres rückgängig zu machen. Mißerfolge können evtl. Strafen nach sich ziehen. Bei der interaktiven Beschäftigung mit dem Rechner hingegen lassen sich Handlungen oftmals widerrufen. Der Lernende darf Fehler machen, ohne bestraft zu werden. Manche Fehler macht ein Lernender sogar absichtlich, um tutorielle Rückmeldungen vom System zu erhalten. Somit kann die Sanktionsfreiheit der Interaktion einen

wichtigen Aspekt für den Lernenden darstellen. Allerdings sind unmittelbares Feedback und kurze Antwortzeiten in diesem Fall von großer Bedeutung, um die Lernenden zu informieren und ihnen eine Chance zur Fehlerkorrektur zu geben. Die nächsten beiden Absätze beschäftigen sich mit zwei Merkmalen der Interaktion, die meist in hypermedialen Lernsystemen Anwendung finden: Metaphern und Navigation.

Metaphern Im Umgang mit hypermedialen Lernsystemen ist die Verwendung von *Metaphern* weit verbreitet. Diese dienen sowohl der räumlichen als auch der zeitlichen Orientierung der Lernenden bei der Benutzung der Systeme. Sie sollen den Systemzugang und das Verständnis der Struktur vereinfachen sowie das mentale Modell des Lernenden mit dem Modell des Systems in Übereinstimmung bringen. Autoren erhoffen sich über diese Korrespondenz eine gewisse Regulation der Interaktion und eine bestimmte Sichtweise auf die Inhalte. Die nachfolgende Liste benennt einige häufig angewandte Metaphern, die in neueren Lernsystemen vorkommen (nach [Sch96, 49 f.], [Blu98, 190]):

- *Buch*: elektronisches Textbuch (vgl. Abschnitt 6.3.1), Lexika
- *Raum*: Warenhaus, Reisen in einer Landschaft
- *Guides*: animierte Begleitpersonen
- *Geschichtenerzähler*: Lerngespräche
- *Zeit*: Zeitleisten
- *Virtuelle Gerätschaften*: Kompaß, Kamera
- *Schnittstellen*: Schreibtisch, Dialoge

Schulmeister und andere Autoren warnen allerdings vor der unbedachten Nutzung von Metaphern. Die Metapher muß mit Inhalt und Sachgebiet übereinstimmen, d. h. nicht jede Metapher ist für jeden Inhalt geeignet.

Navigation Auf einer weiteren Metapher zur Beschreibung der Orientierungs- und Suchaktivitäten der Benutzer von Hypertext- bzw. Hypermediasystemen basiert der Begriff *Navigation*. Darunter fallen auch Entscheidungen eines Lernenden im Verlauf der Nutzung eines hypermedialen Lernsystems. An dieser Stelle beschränken wir uns auf den Navigationsbegriff, der in Lernsystemen gebräuchlich ist: der Benutzerführung. Navigation findet natürlich auch in anderen Bereichen statt, die in dieser Arbeit ebenfalls beschrieben werden, etwa bei der Orientierung in komplexen Softwarevisualisierungen.

Tergan [Ter97, 127 f.] klassifiziert Navigationsprozesse in drei Formen: Browsing, gezielte Suche mit der Hilfe von Suchalgorithmen und Nachgehen von vorab festgelegten Pfaden. Zur Durchführung der Navigation bedient man sich einer Reihe von Werkzeugen (vgl. [Blu98, 191 ff.], [Haa97, 156 f.]):

- *Graphische Übersichten* bilden die Struktur des zugrundeliegenden Systems oder Gegenstands ab, z. B. Karten, Bäume und Graphen.

- *Pfade* bieten vorab festgelegte Wege durch ein Netzwerk von Informationseinheiten an (Guided Tours, Trails).
- *Backtracking* und *History* ermöglichen ein schrittweises Zurückverfolgen des eingeschlagenen Pfades bzw. den direkten Zugriff auf bereits besuchte Informationseinheiten.
- *Lesezeichen* können von Lernenden genutzt werden, um subjektiv wichtige Bereiche zu markieren. Dies erlaubt eine persönliche Orientierung und die Festlegung eigener Prioritäten.
- *Marker* sind automatische Kennzeichen in bereits bearbeiteten Bereichen, z. B. die Hyperlinkfärbung in Webbrowsern.
- *Autorenhinweise* kennzeichnen wichtige Lernbereiche und werden von den Autoren eines Lernsystems vorgegeben.
- *Guides* geben Hilfestellung und Ratschläge für das weitere Vorgehen des Lernenden.
- *Suchfunktionalitäten* werden durch Suchmaschinen o. ä. bereitgestellt.

Die Aufgabe der genannten Navigationswerkzeuge umfaßt die Vermeidung oder Reduzierung der bekannten Probleme, die während der Arbeit mit hypermedialen Lernsystemen auftreten können. Hierbei ist zu berücksichtigen, daß eine wohldurchdachte Konzeption und Strukturierung eines Systems in jedem Fall sinnvoller ist als eine nachträgliche Ausstattung des Systems mit diesen Werkzeugen. Im nächsten Abschnitt betrachten wir einige Problemklassen in Zusammenhang mit Multi- und Hypermedia näher.

7.3.2 Problemfelder

Es existieren eine Reihe von Problemen, die in der medienwissenschaftlichen Literatur immer wieder in Erscheinung treten. Die meisten gründen auf den bekannten Schwierigkeiten mit der Navigation in hypertextbasierten Lernumgebungen und den dort möglichen Orientierungsproblemen. Im folgenden beleuchten wir kurz die drei bekanntesten Grundtypen von Problemen: Segmentierung der Inhalte sowie Desorientierung und Überbelastung des Lernenden.

Segmentierung Ein Problem, welches eng mit der Navigation verknüpft ist, stellt die Segmentierung der einzelnen Lerneinheiten dar. Wie in Abschnitt 7.2.1 erwähnt, werden „die Informationen im computerbasierten Lernen meist in kleine, in sich abgeschlossene Einheiten aufgeteilt“ [Blu98, 187]. Diese Granularität muß jeweils in Abhängigkeit von dem zu lernenden Sachverhalt festgelegt werden. Bei zu kleinen Einheiten kann es zu einem hohen Grad an Atomisierung kommen, der diese Einheiten vom Gesamtkontext des Systems isoliert. Bei der Gestaltung der Navigation versucht man, der Atomisierung entgegenzuwirken.

Desorientierung Ebenfalls eng an die Navigation gebunden ist die These, daß die Orientierung innerhalb von größeren hypermedialen Umgebungen, Simulationen usw. mit großen Schwierigkeiten gekoppelt sein kann. „Lost in Hyperspace“ [Con87] ist eine berühmte Umschreibung der Desorientierung. Lernende haben z. B. oft das Gefühl, wichtige Informationen nicht aufgenommen zu haben. Manche Autoren sehen hier immanente Nachteile großer Lernumgebungen; andere wiederum erkennen, daß jede Art der Exploration natürlicherweise mit einem Defizit an Startinformation verbunden ist. Dieses Defizit muß in Kauf genommen und zu große Desorientierung, welche frustrierend wirkt, durch passende Navigationsmaßnahmen vermieden werden (vgl. [Sch96, 55]).

Desorientierung kann auch dadurch entstehen, daß der Lernende auf der Suche nach bestimmten Informationen das ursprüngliche Ziel seiner Anstrengungen zuweilen vergißt. Andere Informationen rücken in den Mittelpunkt des Interesses (*Serendipity-Effekt*). Kuhlen [Kuh91] argumentiert, daß in diesem Effekt eine Analogie zum entdeckenden Lernen zu finden ist und erachtet diese als positive Erscheinung (s. a. [MKA90]).

Kognitive Überlast Man versteht unter dem Begriff der kognitiven Überlast „die zusätzliche kognitive Belastung, die dadurch entsteht, daß beim Lernen mit Hypertext/Hypermedia zusätzliche Gedächtniskapazität benötigt wird, um die bereits besuchten bzw. noch nicht besuchten Informationsknoten, bestehende Navigationsmöglichkeiten, bereits gebildete mentale Repräsentationen etc. im Gedächtnis zu behalten“ [IK97, 485]. Beispiele hierfür sind die infolge von Desorientierung erhöhte Benutzung von Navigationsmitteln, die Steuerung von Animationen usw. Für den Lernenden offenbaren sich viele verschiedene Handlungsalternativen, aus denen er wählen kann. Neben der altbekannten Forderung nach einem sinnvollen Softwaredesign zur Reduzierung dieses Problems, stellt Blumstengel [Blu98, 187] fest, daß sich die kognitive Belastung nach dem Erwerb einer gewissen Kompetenz im Umgang mit hypermedialen Lernsystemen von selbst verringert.

7.4 Stufen explorativen Lernens

Bei der Entwicklung von Lernsoftware für den Übersetzerbau steht die Visualisierung und Animation von Berechnungsmodellen im Mittelpunkt des Interesses. Dieser Abschnitt untersucht die Fragestellung, wie die Generierung solcher Visualisierungen und die Visualisierung des Generierungsprozesses selbst das Explorationserleben des Lernenden steigern. Dazu werden vier Ansätze mit wachsender Explorationsmöglichkeit in der Theorie formaler Sprachen sowie im Übersetzerbau eingeführt [DK00a, DK01]. Anschließend analysieren wir die in Kapitel 6 präsentierten Anwendungen aus der Sicht dieser Ansätze sowie der vorangegangenen Erörterungen zur Lerntheorie und Multimedia.

7.4.1 Lernsoftware für den Übersetzerbau

Heutige Lernsoftware behandelt größtenteils Themen außerhalb der Informatik und zielt meist auf die Vermittlung von Faktenwissen und weniger auf das Verständnis komplexer

Abläufe ab. Hierzu zählen elektronische Bücher, Lexika und Wörterbücher. Schaut man sich z. B. die von Schulmeister beschriebene Lernsoftware [Sch96] (u. a. 16 interaktive, hypermediale Lernprogramme und 23 tutorielle Programme) genauer an, so stellt man fest, daß nur ein geringer Teil davon komplexe Vorgänge darstellen kann. Dies sind vor allem Systeme im Bereich der Physik. Im Gebiet der Informatik überwiegen Kurse für Programmiersprachen. Nur wenige Systeme behandeln Themen der theoretischen Informatik [BDKW99].

In der Informatik und insbesondere im Übersetzerbau sind Theorie und Algorithmen sehr abstrakt und meist auch komplex. Daher ist ihre visuelle Darstellung eine wichtige Hilfe im Informatikunterricht [Ker97, Ker99]. Obgleich der Übersetzerbau meist als praktisches Gebiet innerhalb der Informatik gesehen wird, beruhen seine Techniken auf Ergebnissen der theoretischen Informatik wie z. B. auf formalen Sprachen, der Automatentheorie und der formalen Semantik.

Häufig tritt im Informatikunterricht das Problem auf, daß der Lehrer ein dynamisches System nur unzureichend auf Papier, Folien oder an der Tafel darstellen kann. Nehmen wir an, daß ein Dozent die Funktionsweise eines Kellerautomaten erläutern will. Bestenfalls stehen ihm dazu eine große Tafel und genügend farbige Kreide zur Verfügung. Nun steht er vor der Herausforderung, den Ablauf des Automaten anhand einer kleinen Beispieleingabe, einer endlichen Zahl von Zuständen und eines Kellerbildes zu erklären. Spätestens nach drei oder vier Schritten beginnt er, Zustände oben auf dem Kellerbild wegzuwischen, neue einzutragen usw. Der Zuhörer ist sehr damit beschäftigt, den komplizierten Ablauf des Wischens und Neuschreibens nachzuvollziehen. Es wäre besser, wenn er sich stärker auf die eigentliche Funktionsweise eines Kellerautomaten konzentrieren könnte. Aufgrund dieser Problematik ist die Präsentation eines solchen Automaten auch nur sehr schwer reproduzierbar. Für diesen Zweck ist die Computeranimation, insbesondere die Algorithmenanimation, das Mittel der Wahl. Mit Hilfe von Computeranimationen lassen sich technische Prozesse anschaulich und intuitiv erklären.

Im Übersetzerbau werden Techniken entwickelt, mit denen Programme höherer Programmiersprachen in effiziente und korrekte Programme einer realen oder abstrakten Maschine übersetzt werden können [WM96]. Die Übersetzung eines Programms kann

Übersetzerphase	**Spezifikationssprache**	**Berechnungsmodell**
Lexikalische Analyse	Reguläre Ausdrücke	Endliche Automaten
Syntaktische Analyse	Kontextfreie Grammatiken	Item-Kellerautomaten
Semantische Analyse	Attributgrammatiken	Attributauswerter
Codeerzeugung	Baumgrammatiken	Baumautomaten
Optimierung	Gleichungssysteme	Fixpunktlöser
Laufzeitsystem	**Spezifikationssprache**	**Berechnungsmodell**
Abstrakte Maschine	Kontrollflußsprache	Maschine mit Kellern, Halde und Registern

Tabelle 7.2: Übersetzerphasen.

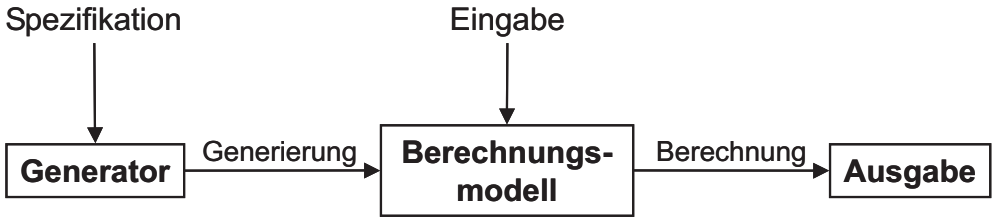


Abbildung 7.1: Zusammenhang zwischen Spezifikation und Generierung.

in verschiedene Phasen unterteilt werden. Eine erste, grobe Strukturierung eines Übersetzungsprozesses ist seine Trennung in eine Analysephase und eine Synthesephase. In der Analysephase werden die syntaktische Struktur und einige semantische Eigenschaften des Quellprogramms berechnet. Idealerweise ist diese Phase von den Eigenschaften der Zielsprache und der Zielmaschine unabhängig. Das Resultat ist ein Syntaxbaum, der mit zusätzlichen Informationen dekoriert ist, z. B. mit dem Typ eines angewandten Vorkommens eines Bezeichners. In der Synthesephase konvertiert der Übersetzer diese Programmrepräsentation in mehreren Schritten in ein äquivalentes Zielprogramm. Dieses wird wiederum auf einer abstrakten Maschine ausgeführt. Abstrakte Maschinen erleichtern die Portierung eines Übersetzers von einer konkreten Maschine auf eine andere sowie die Implementierung von Übersetzern höherer Programmiersprachen, bei denen das zugrundeliegende Berechnungsmodell stark von dem realer Hardwarearchitekturen abweicht, z. B. im Fall von funktionalen oder logischen Programmiersprachen. Aus didaktischer Sicht weisen abstrakte Maschinen noch einen weiteren Vorteil auf: die Entkopplung von speziellen Problemen und Eigenschaften der jeweiligen Rechnerarchitektur (s. a. Abschnitt 6.1.3).

Jede der klassischen Übersetzerphasen kann mit einer Spezifikationssprache beschrieben werden. Mit Hilfe einer solchen Spezifikation wird dann ein Berechnungsmodell, etwa ein deterministischer endlicher Automat, generiert. In Tabelle 7.2 sind die Übersetzerphasen, ihre Spezifikationssprachen und zugrundeliegenden Berechnungsmodelle aufgeführt. Abbildung 7.1 verdeutlicht in einem Diagramm die Abhängigkeiten und Zusammenhänge zwischen Spezifikations- und Generierungsprozeß sowie Berechnungen des generierten Modells. Das Diagramm wird im folgenden wiederholt zur Erläuterung der oben erwähnten Ansätze herangezogen.

7.4.2 Beschreibung des Lernmodells

Bei der Entwicklung von Visualisierungen und Animationen für Berechnungsmodelle bieten sich hinsichtlich ihrer Darstellung und der angestrebten Lernziele mehrere Möglichkeiten. Wir betrachten in dieser Beschreibung unseres Lernmodells der wachsenden Exploration exemplarisch die Übersetzungsphase der lexikalischen Analyse.

Definition 7.1 (Statischer Ansatz) *Im statischen Ansatz wird die Ausführung einer Instanz eines Berechnungsmodells für eine gegebene feste Eingabe animiert.*

Visualisiert wird eine vorgegebene feste Instanz eines Berechnungsmodells, beispielsweise ein deterministischer endlicher Automat A, der die Sprache $L(a^*b^*)$ akzeptiert. Es ist nicht möglich, eine eigene Eingabe zur Simulation einer Berechnung anzugeben, d. h. die Ausführung unseres Automaten erfolgt auf einem ebenfalls vorgegebenen Eingabewort. Falls der Lernende neugierig darauf ist, was mit einem anderen Eingabewort passieren würde, gibt es keinen Weg für ihn, dies herauszufinden. Dieser Ansatz ist folglich stark behaviouristisch geprägt, die Wissensvermittlung über die Arbeitsweise einer speziellen Instanz eines Berechnungsmodells steht im Vordergrund und der eher passive Lernende hat wenig Einfluß darauf, was ihm wie präsentiert wird. Er hat lediglich die Kontrolle darüber, wann die Animation gestartet wird oder wie schnell sie abläuft etc.

Definition 7.2 (Interaktiver Ansatz) *Im interaktiven Ansatz wird die Ausführung einer Instanz eines Berechnungsmodells für eine beliebige benutzerdefinierte Eingabe animiert.*

Dieser Ansatz geht nun einen Schritt weiter. Bleiben wir bei unserem Beispiel des endlichen Automaten A, dann läßt sich in diesem Fall ein beliebiges Eingabewort durch den Lernenden angeben. Die Visualisierung zeigt die infolge dieser interaktiven Eingabe initiierte Berechnung (Simulation) und das ebenfalls vom Eingabewort abhängige Akzeptanzverhalten des Automaten. Der Lerner kann eine Ablehnung eines Wortes, das nicht in $L(a^*b^*)$ enthalten ist, provozieren oder einfach eine Reihe von Eingaben vornehmen, um zu überprüfen, ob diese Worte akzeptiert werden. Das explorative Lernen wird mit diesem Ansatz wesentlich stärker unterstützt. Ein Teil der Animation kann durch den Lernenden selbst gesteuert werden, und zwar nicht nur im Hinblick auf die Kontrollierbarkeit der Animation selbst, sondern auf das, *was* zu sehen ist. Allerdings ist die visualisierte Instanz des Berechnungsmodells immer noch unveränderbar.

Definition 7.3 (Generativer Ansatz erster Ordnung) *Im generativen Ansatz erster Ordnung gibt der Benutzer die Spezifikation einer Instanz eines gegebenen Berechnungsmodells an. Anschließend wird eine interaktive Visualisierung dieser Instanz generiert, und der Benutzer kann analog zum interaktiven Ansatz eine beliebige Beispieleingabe vornehmen.*

Bisher sind Generatoren, die aus einer Spezifikation eine Instanz eines Berechnungsmodells erzeugen, im Lernmodell außen vor gelassen worden. Im generativen Ansatz erster Ordnung kann der Lernende z. B. den regulären Ausdruck a^*b^* als Spezifikation des Automaten A eingeben. Ein entsprechender Generator erzeugt nun daraus eine Implementierung und zusätzlich eine interaktive Visualisierung der Instanz des Berechnungsmodells, in unserem Beispiel den Automaten A. Wie im vorhergehenden Ansatz wird die benutzerdefinierte Eingabe für das Berechnungsmodell unterstützt. Hier soll neben der reinen Wissensvermittlung verstärkt das Verstehen und die Interpretation des Lehrstoffs gefördert werden. Die Information ist nicht durch einen Lehrenden fertig strukturiert worden, sondern Lernende müssen ihre eigenen Informationen entdecken und ordnen. Der Ansatz ermöglicht ihnen, neue Hypothesen zu formulieren und diese

durch Änderung der Spezifikation oder der Benutzereingabe zu überprüfen. Die Manipulierbarkeit der Instanzen über eine Spezifikationssprache führt zu einer Analogie dieses konstruktivistisch geprägten Ansatzes mit dem Mikrowelten-Konzept (siehe Abschnitt 7.2.2.1), welches die Konstruktion von allgemeinen Modellen propagiert, anstatt nur die Arbeit mit einem Modell zu unterstützen. Auf diesem Weg können Lernende viel über das Berechnungsmodell lernen, allerdings nichts über dessen Generierungsprozeß. Dieser wird als „Blackbox“ behandelt.

Definition 7.4 (Generativer Ansatz zweiter Ordnung) *Wie im generativen Ansatz erster Ordnung gibt der Benutzer die Spezifikation einer Instanz eines gegebenen Berechnungsmodells an. Im generativen Ansatz zweiter Ordnung wird jedoch zusätzlich zur Visualisierung der Berechnung auch der Generierungsprozeß selbst in Form einer interaktiven Animation dargestellt.*

Dieser letzte Ansatz, der auch den Generierungsprozeß visualisiert, ermöglicht exploratives, selbstbestimmtes und selbstkontrolliertes Lernen. Der Lerner kann sich auf bestimmte Aspekte in den generierten, interaktiven Animationen konzentrieren und beobachten, welche Auswirkungen kleine Modifikationen der Spezifikation haben. Mit Hilfe solcher Beobachtungen formuliert der Lerner Hypothesen, die er empirisch überprüfen kann. Im interaktiven Ansatz sind derartige überprüfbare Hypothesen durch die Instanz des Berechnungsmodells beschränkt. Im generativen Ansatz erster Ordnung können auch Hypothesen über das Berechnungsmodell und im generativen Ansatz zweiter Ordnung sogar solche über den Generierungsprozeß selbst überprüft werden.

In einer konkreten Lernsituation, die auf diesem Ansatz beruht, beschreibt der Lerner Prozesse durch Spezifikationen als Übungsaufgabe. In konventioneller Lernsoftware werden Antworten, d. h. Lösungen von Übungsaufgaben, auf Korrektheit getestet, wenn dies möglich ist. Auftretende Fehler werden dem Lerner angezeigt und er wird anschließend aufgefordert, seine Antwort zu überarbeiten. In der Informatik können viele Eigenschaften von Berechnungsmodellen aufgrund der Unentscheidbarkeit des Halteproblems nicht überprüft werden. Als Konsequenz brauchen wir alternative Wege, um dem Lerner Feedback zu geben. In diesem Ansatz wird zunächst eine interaktive Animation aus der Antwort (hier also der Spezifikation) des Lerners erzeugt. Dann kann er diese auf der Grundlage eigener Beispiele austesten. So ist es möglich, eigene Fehler aufzudecken. Es gibt keine anonyme, allwissende Autoriät, die auf mögliche Fehler hinweist (Sanktionsfreiheit, vgl. S. 138).

Ganimal wurde entwickelt, um den generativen Ansatz zweiter Ordnung zu unterstützen. Ziel war die Bereitstellung eines allgemeinen Frameworks für die Implementierung von Generatoren, anstatt den Generierungsprozeß eines einzelnen speziellen Berechnungsmodells zu visualisieren. Unser Ansatz ist nicht dazu gedacht, die klassische Lehre theoretischer Inhalte zu ersetzen. Vielmehr soll er diese hilfreich erweitern und unterstützen. Die Akzeptanz und Leistungsfähigkeit einer solchen explorativen Lernsoftware müssen natürlich in der Praxis, d. h. in der Lehre, bewiesen werden. Daher wurden in Zusammenarbeit mit kognitiven Psychologen zwei Evaluationen durchgeführt, die in Kapitel 8 dargestellt werden.

Stufe	Ansatz	Eingabe	Berechnungsmodell	Generator
1	Statisch	Fix	Fix	Nein
2	Interaktiv	Benutzer	Fix	Nein
3	Generativ erster Ordnung	Benutzer	Benutzer	Ja
4	Generativ zweiter Ordnung	Benutzer	Benutzer	Ja, visualisiert

Tabelle 7.3: Vier Stufen explorativen Lernens.

Stufen explorativen Lernens Die oben beschriebenen Ansätze bilden zusammen vier Ebenen oder Stufen, in denen exploratives Lernen in unterschiedlichen Graden stattfinden kann. Tabelle 7.3 reflektiert nicht nur die wachsende Explorationsmöglichkeit durch generative Verfahren, sondern auch die chronologische Entwicklung der in unserer Forschungsgruppe erstellten Lernsysteme. Höhere Explorationsstufen verlangen aber in unserem Lernmodell vom Lernenden mehr Voraussetzungen und Selbstkontrolle. Aus diesem Grund sollte der Lernende bei der Nutzung von Lernsoftware mit den statischen Beispielen beginnen. Dann kann die Explorationsstufe mit seinem eigenen Fortschritt erhöht werden. Übungen und textuelle Hinweise durch das Lernsystem sollten idealerweise sicherstellen, daß der Lernende keine wichtigen Lerneinheiten versäumt. Aus lerntheoretischer Sicht hält sich dieses Lernmodell an die Empfehlung, Mischformen der Instruktion und der lerntheoretischen Fundierung zu bevorzugen (vgl. Abschnitt 7.2.3).

Das vorgestellte Lernmodell macht nicht nur im Bereich des Übersetzerbaus und dessen speziellen Problemstellungen Sinn, sondern ist auch in anderen Disziplinen anwendbar, in denen Generatoren eingesetzt werden.

7.4.3 Fallstudien

In den letzten Jahren haben wir mehrere Lernsysteme für Themen aus dem Übersetzerbau und der theoretischen Informatik entwickelt (vgl. Kapitel 6). Diesen Systemen ist gemeinsam, daß sie Berechnungsmodelle durch animierte Berechungen von Instanzen dieser Modelle auf Beispieleingaben vermitteln. Die Unterschiede liegen in der Explorationsstufe.

7.4.3.1 Statischer Ansatz

Die in Abschnitt 6.1.1 präsentierte Lernsoftware „Animation der lexikalischen Analyse – ADLA“ entspricht dem statischen Ansatz, weil die Ausführung vorgegebener endlicher Automaten als Berechnungsmodell-Instanzen auf ebenfalls fest vorgegebenen Eingaben animiert werden[1]. Lerner können die Animationen starten, anhalten oder diese zurücksetzen, aber keine eigenen Eingaben vornehmen.

[1]Die in Abschnitt 7.4.3 präsentierten Abbildungen 7.2 bis 7.5 verdeutlichen unsere vier Ansätze graphisch. Dabei bezeichnen grau hinterlegte Bildteile *visualisierte* und dick umrahmte Bildteile *erzeugte* Komponenten.

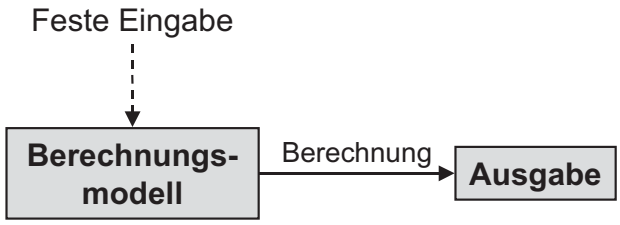


Abbildung 7.2: Stufe 1 – Statischer Ansatz.

Das System wurde mit einem Autorensystem erstellt und folgt daher einem instruktionalistischen Lernparadigma. Es realisiert die Buchmetapher und eine stark lineare Navigation. In bestimmten Lerneinheiten können die o. g. Animationen aufgerufen werden. Insgesamt gesehen ist die Lernerkontrolle auf den Seitenwechsel und den Start der Animationen beschränkt.

7.4.3.2 Interaktiver Ansatz

Ein Beispiel für den interaktiven Ansatz ist die Anwendung „Animation der semantischen Analyse – AdsA“ (vgl. Abschnitt 6.1.2). In dieser Lernsoftware entspricht das Berechnungsmodell der Attributauswertung eines Programms. Die Instanzen korrespondieren mit der jeweils animierten Überprüfung der Kontextbedingungen, der Auflösung der Überladung und der Typinferenz für eine Sprache mit parametrischem Polymorphismus. Obwohl der Benutzer beliebige eigene Beispiele eingeben kann, ist es ihm nur möglich, eine der drei vorgegebenen semantischen Analysemethoden auszuwählen, die dann anhand der Eingabeprogramme bzw. Eingabespezifikationen für die Auflösung der Überladung animiert werden.

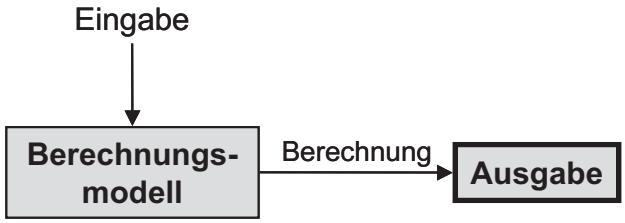


Abbildung 7.3: Stufe 2 – Interaktiver Ansatz.

Für die Entwicklung dieses Lernsystems wurden im statischen Teil dieselben Techniken verwendet wie im AdlA-System. Es setzt ebenfalls die Buchmetapher um und ist sequentiell aufgebaut. Die dynamisch aus den Eingabeprogrammen erzeugten Animationen bieten jedoch ein höheres Maß an Interaktion mit dem System. Verschiedene Dialoge fragen diverse Einstellungen für die Visualisierung ab, der Lerner kann Komponenten der dargestellten Syntaxbäume auswählen und Informationen zu diesen abrufen. Eingebaute Editoren unterstützen den Lernenden bei der Eingabe der Beispielprogramme, finden Syntaxfehler und melden diese.

7.4.3.3 Generativer Ansatz erster Ordnung

Als Beispiel des generativen Ansatzes erster Ordnung betrachten wir GANIMAM, den web-basierten Generator für interaktive Animationen abstrakter Maschinen (vgl. Abschnitt 6.1.3). Das System implementiert den generativen Ansatz erster Ordnung, da der Lerner die Spezifikation einer abstrakten Maschine als Instanz eines gegebenen Maschinenmodells festlegen kann. Ein Generator erzeugt aus dieser Beschreibung eine Visualisierung der spezifizierten Maschine, auf der selbstgeschriebener bzw. aus einem Hochsprachenprogramm automatisch erzeugter Maschinencode ausgeführt werden kann.

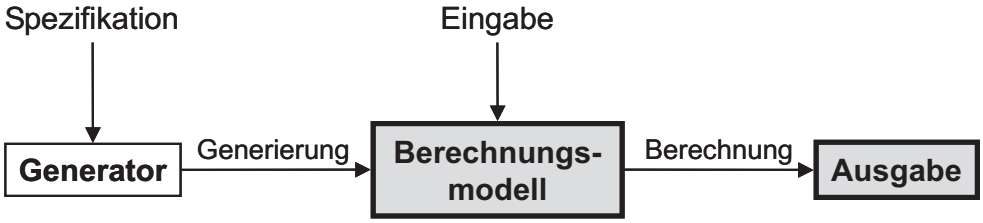


Abbildung 7.4: Stufe 3 – Generativer Ansatz erster Ordnung.

Zur Laufzeit eines Maschinenprogramms bietet das System nicht nur eine für Lernende kontrollierbare Animation der Berechnung, sondern darüber hinaus auch die Möglichkeit, Werte in den Registern oder Speicherzellen zu manipulieren. GANIMAM selbst ist dabei nicht in ein multimediales Lernsystem über abstrakte Maschinen im Übersetzerbau eingebettet, d. h. es existieren keine tiefergehenden Erklärungen im Sinne eines multimedialen Lernsystems. Es ist lediglich eine Dokumentation über die Funktionalität, und insbesondere über die der angebotenen Spezifikationssprache, vorhanden. GANIMAM kann dazu verwendet werden, neue Prinzipien durch Experimentieren mit Maschinenprogrammen oder mit Spezifikationen von abstrakten Maschinen zu entdecken. Aufgrund dieser experimentellen Vorgehensweise ist das System dazu prädestiniert, als Bestandteil einer explorativen, konstruktivistisch orientierten Lernumgebung verwendet zu werden. Ferner kann es auch zum Lösen von Übungsaufgaben im Rahmen einer konventionellen Lehrveranstaltung eingesetzt werden. In den Sommersemestern 2001 und 2002 fanden Vorlesungen an den Universitäten Trier und Saarbrücken statt, die GANIMAM auf diese Art und Weise nutzten.

7.4.3.4 Generativer Ansatz zweiter Ordnung

Als ersten Testfall des GANIMAL-Frameworks und als Anwendung der vierten Stufe explorativen Lernens diente GANIFA, die Implementierung und Visualisierung eines Generators für die lexikalische Analyse. Der Generator wurde in ein Applet zur Visualisierung der Generierung und Berechnung endlicher Automaten integriert, das in einem elektronischen Textbuch über die Theorie endlicher Automaten verwendet wird. GANIFA wurde in Abschnitt 6.3 ausführlich beschrieben und erfüllt aufgrund der möglichen Visualisierung und Animation des Generierungsprozesses die Anforderungen des generativen Ansatzes zweiter Ordnung.

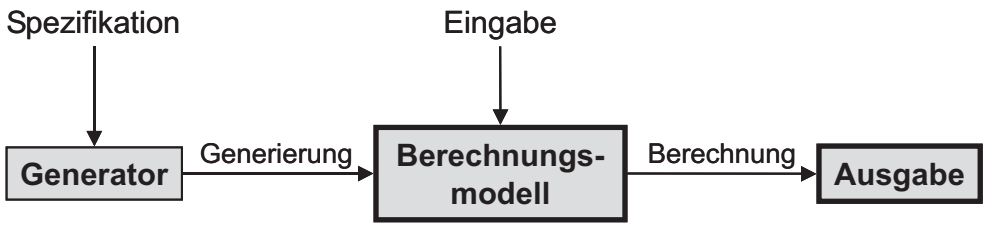


Abbildung 7.5: Stufe 4 – Generativer Ansatz zweiter Ordnung.

Das Textbuch verwirklicht ebenfalls die Buchmetapher, ist aber in HTML geschrieben, basiert also auf einer Hypertextstruktur und ist im WWW frei verfügbar. Die Spezifikation der Automaten erfolgt entweder über Applet-Parameter oder zur Laufzeit über die Eingabe von regulären Ausdrücken. Lerner können einen Teil der verschiedenen Generierungsstadien, d. h. von regulären Ausdrücken über nichtdeterministische endlichen Automaten bis hin zu minimalen deterministischen endlichen Automaten, ausblenden und sich z. B. nur Berechnungen (Simulationen) auf den erzeugten Automaten mit selbstgewählten Eingabeworten anschauen. Obwohl das GaniFA-Applet und unser elektronisches Textbuch nur einen kleinen Teil der Theorie der Generierung endlicher Automaten abdecken, können sie für Einführungskurse sehr nutzbringend sein. Sie eröffnen einen neuen Weg, auf das Lehrmaterial zuzugreifen, und erlauben ein exploratives, selbstbestimmtes Lernen. Lehrer und Lerner können unser Textbuch nicht nur benutzen, sondern sie können darüber hinaus das GaniFA-Applet in ihre eigenen Lehr- bzw. Lernskripte und Übungen einbetten.

8 Evaluation

Unter *Evaluation* versteht man die systematische Anwendung von empirischen Untersuchungsmethoden zur Bewertung einer Maßnahme, eines Sachverhalts oder einer Innovation, wie z. B. einer Technik, einer Methode, eines Systems oder Produkts. In diesem Kapitel wird die Lernwirksamkeit sowie die Akzeptanz und Benutzerfreundlichkeit des elektronischen Textbuchs über die Automatentheorie und des GANIFA-Applets untersucht. Im Mittelpunkt des Interesses liegt hierbei die spezielle Eigenschaft dieses Systems, Animationen aus benutzerdefinierten Spezifikationen generieren zu können.

Eine Evaluation neuer multimedialer Lernsysteme soll über deren Nutzbarkeit und Effizienz Aufschluß geben. Leider werden die meisten neuen Systeme nicht ausreichend evaluiert. Der Grund mag darin liegen, daß die Möglichkeit besteht, nicht nur positives Feedback zu erhalten. Mögliche Folgen einer solchen Unterlassung sind schlechte Benutzbarkeit, Inkonsistenzen in der Darstellung oder praxisfremde Aufbereitung der Lehrinhalte. Begleiten mehrere Evaluationen die Implementierungsphase der Software, man spricht von *formativer Evaluation*, so können im Vorfeld Schwachstellen im Ansatz und der Programmierung sukzessive beseitigt werden. Im Gegensatz dazu gibt eine *summative Evaluation* eine abschließende Beurteilung der „fertiggestellten“ Lernsoftware. Diese wird anhand von möglichst objektiven und quantitativen Daten sowie durch den Einsatz statistischer Methoden vorgenommen (vgl. [Blu98, 174 ff.]). Die in diesem Kapitel präsentierte Evaluation beruht auf dem summativen Ansatz[1].

Allgemein wird ein Nachweis des Gesamtnutzens multimedialer Lernsysteme dadurch erschwert, daß die Systementwicklung sehr teuer ist und im Vergleich zu traditionellen Lernformen nur wenig Erfahrungswerte vorliegen. Hinzu kommt, daß Wechselbeziehungen zwischen Variablen möglichst klein gehalten werden müssen, damit keine falschen Rückschlüsse gezogen werden. Beispielsweise muß unterschieden werden, ob ein überlegener Lernerfolg einer bestimmten multimedialen Lernsoftware darauf zurückzuführen ist, daß diese Software didaktisch anspruchvoller, konstruktiver und interaktiver als ein äquivalentes Buch ist, oder darauf, daß sie als „Neues Medium“ interessanter und dadurch motivierender (sog. *Hawthorne-Effekt*) ist.

Dieses Kapitel ist wie folgt strukturiert: Abschnitt 8.1 diskutiert zwei verschiedene Untersuchungsmethoden, die zur Durchführung einer Evaluation genutzt werden können. Nach einer Einführung in die wichtigsten statistischen Grundlagen in Abschnitt 8.2 stellt Abschnitt 8.3 die Ergebnisse einer Vorstudie zum ADLA-Lernsystem dar, welche

[1] Berücksichtigt man den Einfluß der in Abschnitt 8.3 durchgeführten Vorstudie auf die Entwicklung des elektronischen Textbuchs und des GANIFA-Applets, dann könnte man die Vorstudie auch als Teil einer formativen Evaluation des GANIFA-Lernsystems bezeichnen.

als Erfahrungsgrundlage für die nachfolgende Entwicklung und Evaluation des GaniFA-Systems diente. Schließlich beschreibt Abschnitt 8.4 die Planung, Durchführung und Analyse eines Lernexperiments, in dem GaniFA mit drei weiteren Lehrmitteln verglichen wurde.

8.1 Qualitative vs. quantitative Verfahren

Ein wichtiges Unterscheidungsmerkmal zwischen qualitativen und quantitativen Methoden ist die Art des verwendeten Datenmaterials. Die *qualitative Forschung* operiert auf der Ebene von Interpretationen von verbalem, nichtnumerischem Material wie z. B. Interviewtexten, Beobachtungsprotokollen oder Zeichnungen. Eine Standardisierung des Untersuchungsvorgangs ist meist nicht notwendig. In einer Befragung erhält der Interviewer beispielsweise vollkommen unterschiedliche Antworten, die sich evtl. nicht nur auf die eigentliche Frage beziehen, sondern auch zusätzliche Informationen enthalten. Die *quantitative Forschung* beruht hingegen auf Quantifizierungen von Ausschnitten der Beobachtungsrealität und mündet in die statistische Aufbereitung von Meßwerten. In diesem Fall ist bei einem Interview eine standardisierte Befragung zur Ermittlung numerischen Zahlenmaterials unumgänglich. Die Standardisierung erfolgt z. B. durch den Einsatz sog. Rating-Skalen oder die Angabe von Prozentwerten. Folglich sind die ermittelten Informationen verhältnismäßig leicht auszuwerten und zu vergleichen (vgl. [BD02, 271]).

Beide Ansätze gelten als unvereinbare Gegensätze, obwohl deren Vor- und Nachteile unterschiedlicher Natur sind. So eignen sich qualitative Befragungen eher für redefreudige Personen, wenn persönliche Themenbereiche analysiert werden sollen. Andererseits schafft das Ankreuzen auf einem Fragebogen mehr Distanz zum Forscher und wirkt wesentlich anonymer. Daher werden beide Herangehensweisen in Evaluationen häufig kombiniert (vgl. [BD02, 273]).

Die in diesem Kapitel diskutierte Evaluation basiert hauptsächlich auf der quantitativen Forschung. Einige Punkte des Bewertungsfragebogens, dargestellt in Anhang B.2, untersuchen aber auch qualitative Fragestellungen über die persönliche Einschätzung der eingesetzten Lehrmittel durch die Versuchspersonen.

8.2 Statistische Grundlagen

Zielsetzung dieses Abschnitts ist es, die wichtigsten Begriffe und Formeln kurz zu definieren und sie dem statistisch vorgebildeten Leser wieder in Erinnerung zu rufen. Als ein- bzw. weiterführende Literatur sei beispielsweise auf die Lehrbücher von Bortz [Bor99][2] oder Milton und Arnold [MA86] verwiesen.

Quantitative Informationen sind eine Voraussetzung für die Anwendung statistischer Verfahren. Statistische Analysen können in zwei Teilbereiche untergliedert werden: die

[2]Der Grundlagenabschnitt 8.2 zur Statistik basiert im wesentlichen auf dem Lehrbuch von Bortz.

deskriptive Statistik und die Inferenzstatistik bzw. schließende Statistik. Statistische Methoden zur Beschreibung vorliegender Informationen in Form von Graphiken, Tabellen oder einzelnen Kennwerten werden zusammenfassend als *deskriptive Statistik* bezeichnet. Diese leitet zu einer übersichtlichen und prägnanten Aufbereitung der Daten an und bietet eine gute Übersicht über die in Stichproben angetroffenen Merkmalsverteilungen. Zu beachten ist hierbei, daß über die Beschreibung des erhobenen Datenmaterials hinausgehende Interpretationen bei der deskriptiven Analyse i. allg. spekulativ sind.

Im Unterschied zur deskriptiven Statistik ermöglicht die *Inferenzstatistik* die Überprüfung der Korrektheit von Hypothesen aufgrund empirischer Daten. Mit ihrer Hilfe läßt sich angeben, wie gut aufgrund der Untersuchung einer relativ kleinen Teilmenge (Stichprobe) einer Grundgesamtheit (Population) auf die Verteilung der untersuchten Merkmale aller Einheiten in dieser Grundgesamtheit geschlossen werden kann.

8.2.1 Deskriptive Statistik

Eine Beschreibung der Merkmalsverteilung kann, ausgehend vom empirischen Datenmaterial, tabellarisch in Form von (evtl. kumulierten) Häufigkeits- oder Prozentwertverteilungen erfolgen. Graphische Darstellungen sind beispielsweise Polygone, Histogramme oder Kreisdiagramme. Tabellen und Graphiken informieren über die Verteilung eines Merkmals in einer Stichprobe.

Statistische Kennwerte hingegen geben über spezielle Eigenschaften der Verteilung summarisch eine Auskunft. Zu ihnen gehören die Maße der zentralen Tendenz, die alle Meßwerte gemeinsam repräsentieren, sowie sogenannte Dispersionsmaße, welche die Unterschiedlichkeit in einer Merkmalsausprägung verdeutlichen. Zu den am meisten verwendeten Maßen der zentralen Tendenz gehören die folgenden Werte:

- Der *Modalwert* (Mo) einer Verteilung ist derjenige nichteindeutige Wert, der am häufigsten besetzt ist. Die Verteilung nimmt dort ihr Maximum an.
- Als *Median* (Md) wird der Wert bezeichnet, der eine geordnete Reihe von Beobachtungen in zwei gleich große Teile untergliedert. Betrachtet man eine Häufigkeitsverteilung, dann überschreitet diese die 50 %-Marke an genau diesem Wert (0,5-Quantil).
- Das gebräuchlichste Maß ist das arithmetische Mittel oder der *Mittelwert* ($\overline{x}$) über alle Meßwerte x_i ($1 \leq i \leq n$):

$$\overline{x} = \frac{1}{n} \sum_{i=1}^{n} x_i \tag{8.1}$$

Weitere Maße der zentralen Tendenz sind das geometrische, harmonische und gewichtete arithmetische Mittel, die allerdings seltener angewendet werden. Verteilungen können sich hinsichtlich der o. g. Maße ähneln, aber dennoch auf Grund verschiedener Streuungen stark voneinander abweichen. Die häufigsten Dispersionmaße einer empirischen Verteilung sind nachstehend angegeben:

- Die *Varianz* (s^2) entspricht dem Durchschnitt der quadrierten Differenzen aller n Meßwerte vom arithmetischen Mittel. Die Quadrierung verstärkt den Einfluß größerer Abweichungen auf dieses Maß:

$$s^2 = \frac{1}{n}\sum_{i=1}^{n}(x_i - \overline{x})^2 \tag{8.2}$$

- Da die Varianz wegen der Quadrierung nur schwer zu interpretieren ist, betrachtet man die Wurzel dieses Wertes, die sogenannte *Standardabweichung* (s):

$$s = \sqrt{s^2} = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(x_i - \overline{x})^2} \tag{8.3}$$

Sollen stochastische Zusammenhänge zweier Merkmale x und y untersucht werden, so liefern die folgenden beiden Maße Informationen über die Enge dieser (linearen) Zusammenhänge. Sie können positive und negative Werte annehmen, wobei hohe positive Werte für einen starken gleichsinnigen Zusammenhang, hohe negative Werte für einen starken gegenläufigen Zusammenhang sprechen.

- Die *Kovarianz* wird über den Durchschnitt aller Produkte von korrespondierenden Abweichungen vom jeweiligen Mittelwert definiert. Ein bedeutender Nachteil ist ihre Abhängigkeit vom Maßstab der zugrundeliegenden Variablen bzw. von deren Varianz, d. h. es muß ein verbindlicher Maßstab (z. B. spezielle Zeiteinheiten) vorgegeben werden:

$$\mathrm{cov}(x,y) = \frac{\sum_{i=1}^{n}(x_i - \overline{x})(y_i - \overline{y})}{n} \tag{8.4}$$

- Der *Korrelationskoeffizient* r (Bravais-Pearson-Korrelation) hat diesen Nachteil nicht und ist demzufolge gegenüber Veränderungen im Maßstab der untersuchten Merkmale invariant. Streuungsunterschiede zwischen den Merkmalen werden einfach durch die Division der Kovarianz durch das Produkt der Standardabweichungen beider Merkmale kompensiert. Der Wertebereich dieser Größe liegt im Intervall [-1,1]. Wird eine der Intervallgrenzen als Wert angenommen, dann geht der stochastische Zusammenhang in einen funktionalen, deterministischen Zusammenhang über:

$$r = \frac{\mathrm{cov}(x,y)}{s_x s_y} = \frac{\sum_{i=1}^{n}(x_i - \overline{x})(y_i - \overline{y})}{\sqrt{\sum_{i=1}^{n}(x_i - \overline{x})^2}\sqrt{\sum_{i=1}^{n}(y_i - \overline{y})^2}} \tag{8.5}$$

8.2.2 Schließende Statistik

Statistische Kennwerte können sowohl für Stichproben, als auch für Grundgesamtheiten ermittelt werden. In Grundgesamtheiten (Populationen) werden diese als *Parameter* bezeichnet und in griechischen bzw. in Großbuchstaben dargestellt, z. B. μ als Mittelwert, σ als Standardabweichung oder N als Umfang einer Grundgesamtheit. Stichprobenverteilungen werden wie oben durch Kleinbuchstaben $(\overline{x}, s, n)$ gekennzeichnet. In den meisten empirischen Untersuchungen sind die Populationsparameter allerdings nicht bekannt und müssen aus den Stichprobendaten geschätzt werden. Aus demselben Grund sollte die Stichprobe in der Praxis zufällig gezogen werden.

Die Inferenzstatistik behandelt Ergebnisse von Stichproben in statistischen Untersuchungen wie die Ausgänge eines Zufallsexperiments (siehe [Bor99, 61]), d. h. eine Stichprobe $X = (X_1, \ldots, X_n)$ vom Umfang n ist selbst eine Zufallsvariable mit der Realisation $(x_1, \ldots, x_n)$. Folglich stellen Stichprobenkennwerte wie $\overline{x}$ oder s ebenfalls Realisierungen von Zufallsvariablen $\overline{X}$ bzw. S dar und werden darüber hinaus zur Schätzung der entsprechenden Populationsparameter herangezogen. Ein Gütekriterium für Stichprobenkennwerte zur Parameterschätzung ist die sog. *Erwartungstreue*, die genau dann erfüllt ist, wenn der Erwartungswert der Kennwertverteilung dem zu schätzenden Parameter entspricht. So ist $\overline{X}$ ein erwartungstreuer Schätzer für μ, die Varianz S^2 aber nicht, da sie die Populationsvarianz σ^2 um den Faktor $(n-1)/n$ unterschätzt. Folglich erhält man eine erwartungstreue Schätzung dieses Parameters durch die Multiplikation der Varianz mit dem reziproken Wert $n/(n-1)$:

$$\hat{\sigma}^2 = \frac{1}{n-1} \sum_{i=1}^{n} (x_i - \overline{x})^2 = s^2 \frac{n}{n-1} \tag{8.6}$$

Um zu entscheiden, ob ein bestimmter, stichprobenabhängiger Kennwert zu einer präzisen Schätzung führt, berechnet man die Streuung der Stichprobenkennwerteverteilung. Diese wird als *Standardfehler* bezeichnet. Je kleiner der Standardfehler eines Kennwerts ist, desto genauer fällt die Schätzung des entsprechenden Populationsparameters aus. Der Standardfehler des Mittelwerts kann wie folgt mit Hilfe von $\hat{\sigma}^2$ geschätzt werden:

$$\hat{\sigma}_{\overline{x}} = \sqrt{\frac{\hat{\sigma}^2}{n}} \tag{8.7}$$

Gleiches gilt für den Standardfehler der Standardabweichung s, wobei zu beachten ist, daß weder s noch $\hat{\sigma}$ erwartungstreue Schätzungen für den Populationsparameter σ darstellen:

$$\hat{\sigma}_s = \sqrt{\frac{\hat{\sigma}^2}{2n}} \tag{8.8}$$

Signifikanztests

Im vorangegangenen Abschnitt wurden Methoden vorgestellt, die es erlauben, Schlußfolgerungen von den in einer Stichprobe enthaltenen Daten auf Eigenschaften der Grundgesamtheit zu ziehen. In der Testtheorie werden umgekehrt erst Eigenschaften einer Grundgesamtheit postuliert (z. B. besserer Lernerfolg durch eine bestimmte Lehrmethode). Anschließend wird überprüft, inwieweit die postulierten Eigenschaften durch die Untersuchung einer Stichprobe aus der Grundgesamtheit bestätigt werden können [MA86, Bor99].

Aussagen, die den bisherigen Wissensstand erweitern oder mit anderen Theorien in Widerspruch stehen, werden als *Gegen- oder Alternativhypothesen* (Abk. H_1) bezeichnet. Alternativhypothesen beschreiben genau die Theorien, deren Korrektheit durch eine Teststatistik überprüft werden soll. In Abhängigkeit von einer Alternativhypothese wird die sogenannte *Nullhypothese* (Abk. H_0) formuliert, mit der behauptet wird, daß die zur Alternativhypothese komplementäre Aussage richtig sei. Bezogen auf den Vergleich zweier Mittelwerte lassen sich die folgenden drei Hypothesenpaare aufstellen:

$$(1)\ H_1 : \mu_0 > \mu_1 \qquad H_0 : \mu_0 \leq \mu_1$$

$$(2)\ H_1 : \mu_0 < \mu_1 \qquad H_0 : \mu_0 \geq \mu_1$$

$$(3)\ H_1 : \mu_0 \neq \mu_1 \qquad H_0 : \mu_0 = \mu_1$$

Die Entscheidung, welche Hypothese die richtige ist, wird nun dadurch erschwert, daß sich das Untersuchungsergebnis nur auf die Stichprobe bezieht. Es ist nicht auszuschließen, daß die Alternativhypothese aufgrund der Stichprobenauswahl durch einen Test bestätigt wird, obwohl tatsächlich die Nullhypothese auf die Grundgesamtheit zutrifft. Diese Situation wird in Tabelle 8.1 verdeutlicht.

		In der Grundgesamtheit gilt	
		H_0	H_1
Entscheidung auf Stichprobenbasis zugunsten:	H_0	richtige Entscheidung	β-Fehler (Fehler 2. Art)
	H_1	α-Fehler (Fehler 1. Art)	richtige Entscheidung

Tabelle 8.1: α- und β-Fehler bei statistischen Entscheidungen (entnommen aus [Bor99, 107])

Die prinzipielle Vorgehensweise, eine Entscheidung durch einen statistischen Test herbeizuführen, geht von der Annahme aus, daß die Nullhypothese H_0 richtig ist. Nur wenn H_0 mit der in den erhobenen Daten erfaßten Realität „praktisch“ nicht zu vereinbaren ist, darf sie zugunsten von H_1 verworfen werden [Bor99, 107]. Die bedingte Wahrscheinlichkeit, mit der das Ergebnis unter H_0 eintritt, entspricht der Wahrscheinlichkeit für einen Fehler 1. Art (Irrtumswahrscheinlichkeit). Diese muß in Kauf genommen werden, wenn man H_0 aufgrund des Untersuchungsergebnisses verwirft. Beträgt nun die Wahrscheinlichkeit des gefundenen Ergebnisses unter H_0 höchstens 5 %, so wird das Ergebnis als *signifikant* bezeichnet. Beträgt diese Wahrscheinlichkeit nicht mehr als 1 %, so ist das

Ergebnis sehr signifikant[3]. Dieses Signifikanzniveau wird vor Beginn der Untersuchung festgelegt. Auf der anderen Seite ist ein nicht-signifikantes Ergebnis auch kein Beleg für die Richtigkeit von H_0.

In der Praxis kommen meist Testverfahren zur Überprüfung der Unterschiedlichkeit zweier (oder mehrerer) Stichprobenergebnisse zur Anwendung, da Populationsparameter im Regelfall unbekannt sind. In Abhängigkeit von der zu belegenden Theorie existiert eine Vielzahl verschiedener Signifikanztests. Oft werden diese nach der Art der zugrundeliegenden Hypothesen unterschieden: in Tests zur Überprüfung von Unterschiedshypothesen (z. B. t-Test für den Vergleich zweier Mittelwerte aus unabhängigen Stichproben oder F-Test für den Vergleich zweier Stichprobenvarianzen) und Tests zur Untersuchung von Zusammenhangshypothesen (Korrelationsrechnung). Eine Erläuterung der verschiedenen Testarten würde den Rahmen dieser Arbeit deutlich überschreiten. Zum Verständnis der in Abschnitt 8.3 diskutierten Vorstudie reicht das bisher zur Testtheorie Erläuterte aus, da lediglich zwei Untersuchungsgruppen vorhanden waren und ein t-Test (s. o.) durchgeführt wurde. Komplexere Fragestellungen, welche das Zusammenwirken und die Möglichkeit wechselseitiger Beeinflussung mehrerer Variablen berücksichtigen, sind mit diesen Mitteln nicht mehr ohne weiteres lösbar. Werden z. B. im einfachsten Fall mehr als zwei Untersuchungsgruppen auf Mittelwertvergleiche hin untersucht, wie es in Abschnitt 8.4 zur Evaluation von GaniFA der Fall ist, so ist es naheliegend zu vermuten, daß dies mit mehreren t-Tests durchführbar sei. Dazu wären aber $\binom{p}{2}$ Tests nötig, wenn p die Gruppenanzahl beschreibt. Werden nun viele Tests durchgeführt, dann können einige dieser Tests zufällig signifikant werden. Die Wahrscheinlichkeit, mit der dies geschehen kann, entspricht per definitionem genau der Irrtumswahrscheinlichkeit, d. h. für eine solches Mehrfach-Testen müßte die Irrtumswahrscheinlichkeit aufwendig angepaßt werden.

Varianzanalytische Methoden überprüfen die Auswirkungen ein oder mehrerer (unabhängiger) Variablen auf ein bestimmtes Merkmal (abhängige Variable). Beispielsweise entpricht in der Evaluation von GaniFA der zu untersuchende Lernerfolg der abhängigen Variable und die vier Lehrmethoden einer vierfach gestuften, unabhängigen Variable, auch Faktor oder Treatment genannt. Hierbei geht das Verfahren von dem Ansatz aus, daß eine durch die Gesamtvarianz aller Meßwerte quantifizierte Unterschiedlichkeit in den Leistungen der Versuchspersonen (kurz VP) registriert wird. Es wird untersucht, bis zu welchem Grad die Gesamtunterschiedlichkeit auf die vier verschiedenen Lehrmethoden zurückgeführt werden kann. Ist dieser Anteil hinreichend groß, so wird die Nullhypothese zugunsten der Alternativhypothese verworfen, d. h. die Lehrmethoden führen zu signifikant unterschiedlichen Lernerfolgen.

8.3 Vorstudie zum Lernsystem ADLA

Zunächst wurde im Jahr 1999 ein Lernexperiment mit der älteren Lernsoftware AdLA durchgeführt, um einerseits Erfahrungen mit empirischen Experimenten zu sam-

[3] In der Ergebnisdarstellung eines Tests werden signifikante Testgrößen mit *, sehr signifikante Testgrößen mit ** gekennzeichnet.

meln und andererseits eine bessere Grundlage für die didaktische Gestaltung der mit dem GANIMAL-Framework zu erstellenden Systeme zu schaffen. Dafür kooperierte unsere Forschungsgruppe mit Mitarbeitern und Studenten des Lehrstuhls für kognitive Modellierung und Methodologie des Fachbereichs Psychologie an der Universität des Saarlandes (UdS). Gemeinsam wurde ein Experiment vorbereitet, um statistisch korrekt auswertbare Daten über die Lernsoftware zu erhalten. Zielgruppen für das ADLA-System waren Schüler, die in ihren Schulen einen Informatikkurs belegten, sowie Studenten der Informatik. Leider konnten trotz der Bereitschaft vieler saarländischer Informatiklehrer zur Mitarbeit nicht genug Schüler für das Experiment gewonnen werden. Daher verwirklichten wir das Experiment im SS 99 mit Informatikstudenten des zweiten Semesters, um eine nach statistischen Kriterien ausreichende Teilnehmeranzahl zu erreichen.

Trotz einer Aufwandsentschädigung konnte die Anzahl der Versuchspersonen nicht wesentlich gesteigert werden: Es nahmen lediglich 15 Informatikstudenten an diesem Lernexperiment teil. Um mit hoher Wahrscheinlichkeit statistisch signifikante Daten zu erhalten, sind aber mindestens 30 Studenten nötig. Daher war schon zum Zeitpunkt der Datenerhebung ein großes Risiko vorhanden, daß eine spätere Auswertung zu keinen signifikanten Ergebnissen führen könnte.

8.3.1 Einschränkung des Lernstoffs

Sowohl in dieser Vorstudie als auch in der späteren Evaluation des GANIFA-Systems hatten mehrere Gruppen von Versuchspersonen Gelegenheit, sich in einer vorgegeben Zeit von ca. 30 Minuten ein eng umgrenztes Teilgebiet der lexikalischen Analyse mit Hilfe verschiedener Lehrmethoden zu erarbeiten. Das Teilgebiet umfaßte eine allgemeine Übersicht über formale Sprachen, reguläre Sprachen und reguläre Ausdrücke, Übergangsdiagramme, nichtdeterministische und deterministische endliche Automaten sowie die Generierung nichtdeterministischer endlicher Automaten aus regulären Ausdrücken (Algorithmus: $RA \rightarrow NEA$ aus [WM96]).

8.3.2 Experimenteller Ablauf

Die Vorstudie basiert auf einem Leistungsvergleich zwischen zwei Gruppen von VP. Die erste Gruppe (1) lernte das Teilgebiet mit Hilfe der Software, die zweite Gruppe (2) das gleiche Teilgebiet mit Hilfe eines äquivalenten Lehrtextes. Um unverzerrte Schlüsse aus den Leistungsunterschieden ziehen zu können, unterschied sich der speziell für das Experiment erstellte Lehrtext inhaltlich nicht von der Lernsoftware. Alle für ADLA möglichen, positiven Effekte lassen sich aus diesem Grund auf die besseren Präsentationsmöglichkeiten und die Animationen des Lernsystems zurückführen.

Nach einer 30-minütigen Lernphase erhielten beide Gruppen einen identischen Testbogen mit 19 inhaltlichen Testfragen. Die Softwaregruppe bekam zusätzlich einen speziellen Fragebogen zur Anwendbarkeit und Bewertung des Systems, z. B. enthielt dieser Fragen zum Seitenlayout oder zur Animationssteuerung. Der Test bestand aus 9 Wissensfragen, deren Beantwortung primär die Erinnerung und Reproduktion explizit genannter

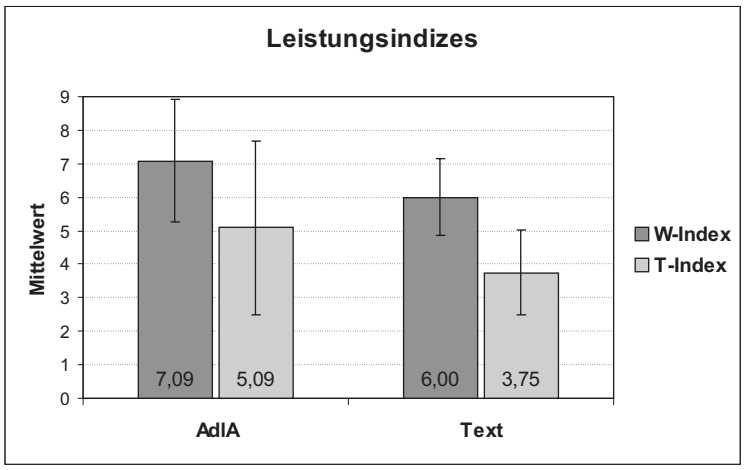


Abbildung 8.1: Deskriptive Statistik der Leistungsindizes.

Sachverhalte forderte, sowie aus 10 Transferfragen, deren Beantwortung von der Versuchsperson ein tieferes Verständnis des Lerninhalts abverlangte. Diese beiden Untertests resultieren in der Auswertung durch die ungewichtete Summierung der richtigen Antworten zu zwei Indizes: einem W-Index für die Wissensfragen und einem T-Index für die Transferfragen.

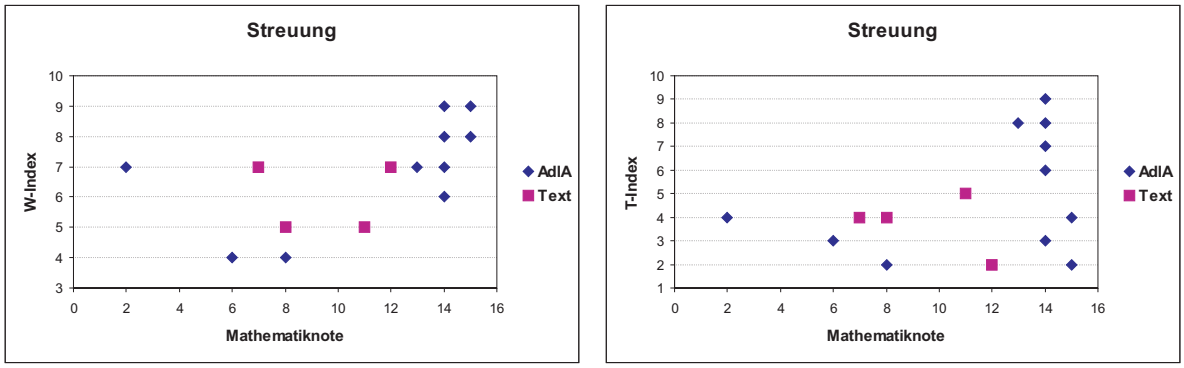
8.3.3 Auswertung

Diagramm 8.1 ist zu entnehmen, daß der Leistungsmittelwert $\overline{x}_1$ der Softwaregruppe für beide Indizes höher ist als der Leistungsmittelwert $\overline{x}_2$ der Textgruppe. Betrachten wir Signifikanzteststudien für den indexabhängigen Vergleich zwischen den Leistungsmittelwerten beider Gruppen (t-Tests), dann lautet die Testhypothese für jeden Index $i \in \{T, W\}$:

$$\begin{aligned} \mathrm{H}_0 : \mu_{1,i} &\leq \mu_{2,i} \\ \mathrm{H}_1 : \mu_{1,i} &> \mu_{2,i} \end{aligned}$$

Nach der Berechnung der Tests betrug der α-Fehler 14.4 % für den W-Index und 10.35 % für den T-Index. Um den Leistungsunterschied zugunsten der Lernsoftware als signifikant deuten zu können, dürfen wir aber das kritische Signifikanzniveau von 5 % nicht überschreiten, d. h. die Alternativhypothese H_1 ist jeweils eindeutig zu verwerfen. Die Irrtumswahrscheinlichkeit hängt allerdings rechnerisch von der Stichprobengröße ab. Da nur 15 Versuchspersonen an der Vorstudie teilgenommen haben, kann davon ausgegangen werden, daß bei einer Wiederholung des Experiments mit mind. 30 Versuchspersonen ein ausreichend großer Datensatz vorläge, um eine statistisch abgesicherte Aussage über die in der deskriptiven Analyse festgestellten Leistungsunterschiede zu machen.

Die deutliche, absolute Mittelwertdifferenz zwischen den Leistungen beider Gruppen scheint allerdings „praktisch bedeutsam“ zu sein. Ein Unterschied, der zwischen zwei



(a) W-Index (b) T-Index

Abbildung 8.2: Streudiagramme für die letzte Mathematiknote in der Schule und der beiden Indizes.

Grundgesamtheiten (hier: Gruppe 1 und 2) mindestens bestehen muß, um von einem praktisch bedeutsamen Effekt reden zu können, wird als *Effektgröße* bezeichnet [Bor99, 115]. Mit ihr wird also festgelegt, wie stark der H_1-Parameter $\mu_{1,i}$ mindestens von $\mu_{2,i}$ abweichen muß, damit der Unterschied als praktisch bedeutungsvoll angesehen werden kann. Für die Berechnung einer Effektgröße wird die Mittelwertdifferenz an der Standardabweichung des untersuchten Merkmals relativiert. Hier definiert sich die Berechnungsvorschrift der effektiv nachgewiesenen Effektgröße ε für jeden Index i wie folgt:

$$\varepsilon_i = \left| \frac{\overline{x}_{1,i} - \overline{x}_{2,i}}{\hat{\sigma}_i} \right| \quad \text{für } i \in \{T, W\} \tag{8.9}$$

Die resultierenden Effekte sind $\varepsilon_W \approx 0.62$ für den W-Index und $\varepsilon_T \approx 0.54$ für den T-Index. Nach gängiger Konvention in den Sozialwissenschaften wird ein Effekt *mittel* genannt, wenn $0.5 \leq \varepsilon < 0.8$, und *stark*, wenn $\varepsilon \geq 0.8$ [Bor99, BD02] ist. Gemessen an diesen Richtwerten ist die festgestellte Leistungssteigerung durch die Lernsoftware durchaus bemerkenswert, kann aber aufgrund der hohen α-Fehler der beiden t-Tests zufällig zustande gekommen sein.

Eine genauere Betrachtung des Datensatzes offenbart weitere Einflußfaktoren auf die Leistungen der VP. Hier stechen Antworten auf die Frage nach der letzten Mathematiknote in der Schule hervor, wie die beiden Streudiagramme in Abbildung 8.2 zeigen. Insbesondere beim W-Index ist der Einfluß der Note gut zu erkennen. Abgesehen von einem Ausreißerwert ergeben sich bei schlechteren Noten tendentiell niedrigere W-Index-Werte als bei guten Noten. Ein linearer Zusammenhang wäre mit einer größeren Stichprobe durchaus zu erwarten. Die Streuung zwischen Mathematiknote und T-Index ist hingegen anders zu interpretieren. Liegen gute Noten vor, dann streuen die T-Index-Werte über die gesamte Y-Achse, während VP mit weniger als 13 Punkten in Mathematik generell nicht mehr als fünf T-Index-Punkte erreichen. Möglicherweise sind gute Mathematikkenntnisse bzw. mathematische Begabung eine notwendige Voraussetzung, aber

keine Garantie für hohe T-Index-Werte. Weniger mathematisch begabte VP wären somit — falls diese Interpretation zutrifft — unabhängig von der Lernsituation mit dieser Art von Testaufgaben überfordert.

8.4 Evaluation von GANIFA

Nach Fertigstellung des GaniFA-Applets und dessen Integration in das elektronische Textbuch wurde im November 2000 ein weiteres Lernexperiment durchgeführt. Als Versuchspersonen wurden 118 Studenten der Informatik ausgewählt, die die Vorlesung „Automaten, Berechenbarkeit und Komplexität“ (Info III an der UdS) besuchten. Es handelte sich dabei hauptsächlich um Informatikstudenten des dritten Semesters. Im Gegensatz zur Vorstudie mit dem älteren AdlA-Lernsystem stand eine ausreichend große Anzahl von Versuchspersonen zu Verfügung. Daher konnten sogar mehrere Lernmethoden in einem Experiment miteinander verglichen werden. Darunter fielen neben dem elektronischen Textbuch GaniFA auch das AdlA-System, ein Lehrtext sowie eine Vorlesung. Im Mittelpunkt der Untersuchung stand aber primär die Frage, ob GaniFA mit seinen generierten Visualisierungen und Animationen ein effizienteres Lehr- und Lernmittel darstellt als beispielsweise ein entsprechender Lehrtext.

8.4.1 Experimenteller Ablauf

Im Vorfeld der eigentlichen Evaluation wurde ein sogenannter Usability-Test mit zwei Psychologiestudenten durchgeführt, um grobe Fehler des elektronischen Textbuchs aufzudecken. Tote Links, Tippfehler oder vergessene Erklärungen zur Bedienung des eingebetteten Applets sind unnötige Fehlerquellen, welche eine Evaluation schon im voraus sinnlos werden lassen.

Insgesamt wurden vier Gruppen gebildet, auf die die 118 Versuchspersonen zufällig verteilt wurden. In diesen Gruppen wurde das in Abschnitt 8.3.1 vorgestellte Teilgebiet der lexikalischen Analyse mit jeweils verschiedenen Lehr- und Lernmethoden vermittelt:

- in Gruppe 1 durch AdlA ($n = 29$)
- in Gruppe 2 durch GaniFA ($n = 30$)
- in Gruppe 3 durch einen Text ($n = 30$)
- in Gruppe 4 durch eine Vorlesung ($n = 29$)

In jeder Gruppe hatten die Versuchspersonen ca. 30 Minuten Zeit, sich mit dem Thema vertraut zu machen. Dabei wurde jeder Versuchsperson in den Softwaregruppen GaniFA und AdlA jeweils ein Computer zugeteilt, an dem sie mit der entsprechenden Lernsoftware arbeiten konnte. Die Versuchspersonen der dritten Gruppe erhielten einen äquivalenten Text, mit dem sie lernen konnten, während die Versuchspersonen der

vierten Gruppe eine Vorlesung besuchten. In jeder Gruppe wurde der identische Lehrinhalt vermittelt und Vorsorge getroffen, daß sich dessen Präsentation in den Gruppen möglichst wenig unterschied. Beispielsweise wurden überall die gleichen Graphiken und Diagramme gezeigt, und die Hörer der Vorlesung durften dem Dozenten, der sich streng an den Lehrtext halten mußte, keine Fragen stellen. Leistungdifferenzen der Gruppen lassen sich daher nur auf die verschiedenen Präsentationen zurückführen. Um diese Unterschiede messen zu können, wurde in allen Gruppen im Anschluß an die Lernphase eine Klausur gestellt. Sie entsprach dem Leistungstest, der schon in der Vorstudie eingesetzt worden war und aus neun Wissens- und zehn Transferfragen bestand. In Anhang B.1 sind diese Fragen zusammen mit einer summativen Häufigkeitsdarstellung der Antworten für jede Untersuchungsgruppe aufgelistet.

Nach der Klausur füllten die Versuchspersonen sowohl einen Fragebogen zur Beurteilung des entsprechenden Mediums, mit dem sie gelernt hatten (Anhang B.2), als auch einen Fragebogen zu ihrer eigenen Person (Anhang B.3) aus. Mit dem Fragebogen zur Bewertung des Lehrmittels konnte gemessen werden, wie unterschiedlich gut die vier verschiedenen Präsentationsmöglichkeiten beurteilt wurden. Der Fragebogen zur Person half, andere Ursachen (z. B. die letzte Mathematiknote in der Schule) auszuschließen, die zu möglichen Effekten hätten führen können.

8.4.2 Ergebnis des Lernexperiments

Die Ergebnisse der Auswertung aller Fragebögen sind in Anhang B tabellarisch dargestellt. Hier sollen die wesentlichen Punkte zusammengefaßt und Schlußfolgerungen abgeleitet werden.

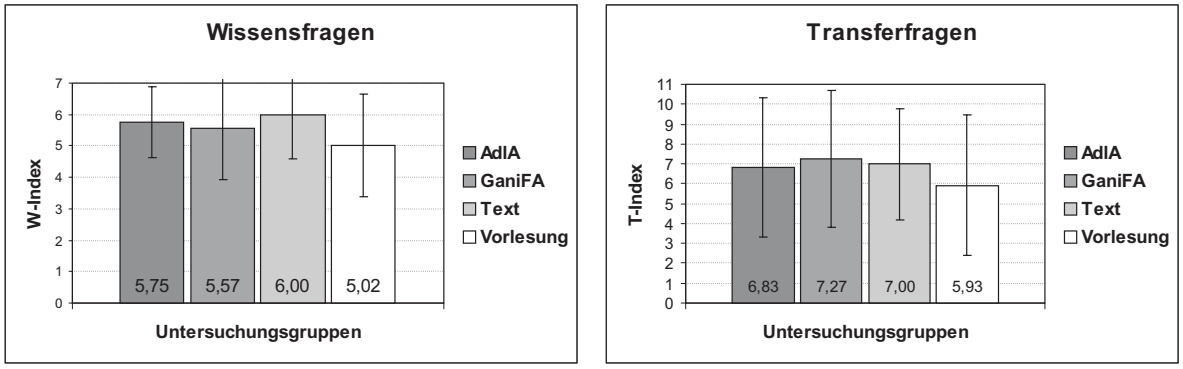
Deskriptive Analyse Durch Aufsummierung der Punkte für die Antworten der im Leistungstest gestellten Wissens- und Transferfragen ergeben sich die in Tabelle 8.2 präsentierten Stichprobenkennwerte. Neben den jeweiligen Gruppengrößen, Mittelwerten und Standardabweichungen sind auch die Standardfehler für die Mittelwerte und die Standardabweichungen angegeben. Sie geben einen groben Eindruck darüber, wie genau die „wahren“ Parameter geschätzt werden.

Um die Leistungsunterschiede noch mehr zu verdeutlichen, sind diese in einem Histogramm für jeden Index (Abb. 8.3) veranschaulicht. Für den W-Index (Maximalwert: 7) zeigt sich, daß der Leistungsmittelwert $\overline{x}_{3,W}$ der Textgruppe am höchsten ist. Für den T-Index (Maximalwert: 13) hingegen ist der Leistungsmittelwert $\overline{x}_{2,T}$ der Gruppe am höchsten, die sich den Lernstoff mit Hilfe des elektronischen Textbuchs GaniFA angeeignet hat. Dieses Ergebnis würde bei Signifikanz eines entsprechenden Tests die Vermutung untermauern, daß ein traditioneller Lehrtext besser zur Vermittlung von Basiswissen geeignet sei als eines der beiden Lernsysteme. Wogegen die Lernsysteme ihre Vorteile in der Vertiefung und Festigung des konventionell erlernten Wissens hätten.

Hypothesenprüfung Eine einfaktorielle Varianzanalyse, welche die Unterschiede des Lernerfolgs zwischen den vier Gruppen auf Zufälligkeit untersucht, erbrachte allerdings

	Untersuchungsgruppe	n	$\overline{x}$	s	$\hat{\sigma}_{\overline{x}}$	$\hat{\sigma}_s$
Wissensfragen	1 – AdlA	29	5.75	1.12	0.21	0.15
	2 – GaniFA	30	5.57	1.65	0.30	0.21
	3 – Text	30	6.00	1.43	0.26	0.18
	4 – Vorlesung	29	5.02	1.65	0.31	0.22
Transferfragen	1 – AdlA	29	6.83	3.48	0.65	0.46
	2 – GaniFA	30	7.27	3.45	0.63	0.46
	3 – Text	30	7.00	2.79	0.51	0.36
	4 – Vorlesung	29	5.93	3.56	0.66	0.47

Tabelle 8.2: Deskriptive Statistik der Leistungsindizes.



(a) W-Index

(b) T-Index

Abbildung 8.3: Histogramme der Indexmittelwerte. Die jeweiligen Standardabweichungen werden als Fehlerbalken repräsentiert.

Untersuchungsgruppe	W-Index	T-Index
1 – AdlA	0.09	0.09
3 – Text	0.27	0.25
4 – Vorlesung	0.40	0.32

Tabelle 8.3: Effektgrößen für den Vergleich von GaniFA (Gruppe 2) mit den übrigen Gruppen.

weder beim W-Index noch beim T-Index signifikante Ergebnisse. Es konnte somit nicht nachgewiesen werden, daß sich die vier Gruppen in ihren Leistungen signifikant unterscheiden. Die Versuchspersonen zeigten, unabhängig von der Art und Weise wie sie lernten, in der Klausur ähnliche Leistungen. Die weiter oben diskutierten Leistungsunterschiede können daher lediglich aufgrund der Stichprobe zufällig eingetreten sein.

Einen Vergleich von Gruppe 2 mit den anderen drei Gruppen hinsichtlich der Effektgrößen zeigt Tabelle 8.3. Die jeweiligen Effekte wurden nach Gleichung (8.9) ermittelt. Gemessen an den auf Seite 160 genannten Richtwerten unterscheidet sich die GaniFA-Gruppe auch nach diesem Kriterium nicht wesentlich von den anderen Gruppen (alle Effekte sind kleiner als 0.5). In unserem Lernexperiment können diese nicht-signifikanten Resultate auf zwei mögliche Ursachen zurückgeführt werden:

- Die vier verschiedenen Lehrmethoden (Treatmentfaktor) üben tatsächlich keinen Einfluß auf die abhängige Variable, d. h. den Lernerfolg, aus oder
- die Fehlervarianz ist im Vergleich zur Faktorwirkung zu groß.

Die Fehlervarianz wird durch unsystematische Effekte nicht kontrollierter Störvariablen erzeugt. Um die Präzision der Analyse zu verbessern, muß der Einfluß derartiger Variablen möglichst klein gehalten werden (vgl. [Bor99]). Eine hierfür oft angewandte Technik ist die sogenannte Kovarianzanalyse (eine Varianzanalyse über Regressionsresiduen), welche die Störvariablen kontrolliert. Voraussetzung für eine effektive Reduzierung der Fehlervarianz durch die Berücksichtigung derartiger Kontrollvariablen ist, daß die abhängige Variable und die Kontrollvariable miteinander korrelieren. Um aber sicherzugehen, daß eine Reduktion kein zufälliges Ergebnis darstellt, sollte ein entsprechender Signifikanztest diese Korrelation absichern.

Bereits in der Vorstudie wurde festgestellt, daß die Leistungen in den Wissens- und Transferfragen von weiteren Einflußfaktoren abhängen (vgl. Abschnitt 8.3). Bei der genaueren Überprüfung des neuen Datensatzes durch mehrere Korrelationsrechnungen ergaben sich lineare Zusammenhänge zwischen den Leistungsindizes und den Variablen S7 bzw. S13–S15, dargestellt in Tabelle 8.4. Die Tabelle zeigt, daß alle Korrelationskoeffizienten bis auf eine Ausnahme auf dem 1 %-Niveau signifikant sind, d. h. die Voraussetzungen für sinnvolle Kovarianzanalysen mit diesen Variablen sind gegeben.

Mehrere Kovarianzanalysen unter Einbeziehung dieser Variablen als jeweilige Kontrollvariablen lieferten die folgenden Ergebnisse: Die Gruppen unterschieden sich signifikant in ihren Leistungen bzgl. der *Wissensfragen*, wenn die Variable S13 „Mathematik als Leistungs- oder Grundkurs“ mit in die Berechnung einbezogen wurde. Hier schnitten vor allem die Textgruppe und die GaniFA-Gruppe besser als die Vorlesungsgruppe und die AdlA-Gruppe ab. Ebenso zeigten die Gruppen auch unter Einbeziehung der Variablen S15 „Informatik als Leistungs- bzw. Grundkurs, freiwillige Arbeitsgemeinschaft oder kein Kurs“ verschiedene Leistungen in den Wissensfragen. Wieder schnitten die Textgruppe und die GaniFA-Gruppe am besten ab. Tabelle 8.5 und Abbildung 8.4 veranschaulichen diese Differenzen.

Trotz der hoch signifikanten Korrelation aus Tab. 8.4 zwischen der Variablen S14 und den Leistungen in den Wissens- und Transferfragen überdeckt die Variable keine

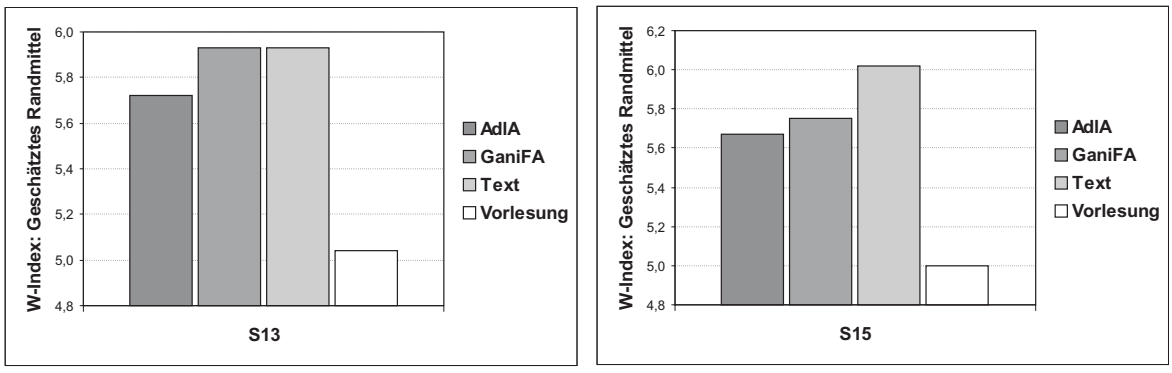
Variablen		W-Index		T-Index	
		r bzw. r_s	α-Fehler	r_s	α-Fehler
S7	Alter, ab dem man Zugang zum ersten Computer hatte.	-0.271**	0.4 %		
S13	Mathematik als Leistungs- oder Grundkurs.	-0.295**	0.1 %	-0.225*	1.6 %
S14	Schulnote, mit der man Mathematik abgeschlossen hatte.	-0.373**	< 0.1 %	-0.286**	0.2 %
S15	Informatik als Leistungs- bzw. Grundkurs, freiwillige Arbeitsgemeinschaft oder kein Kurs.	-0.576**	< 0.1 %	-0.131**	0.1 %

Das Merkmal S7 ist intervallskaliert, d. h. der klassische Bravais-Pearson-Korrelationskoeffizient r kann aus diesem Merkmal und jeweils einem der beiden ebenfalls intervallskalierten Leistungsindizes berechnet werden. Die verbleibenden drei Merkmale (S13–S15) sind allerdings ordinalskaliert, so daß die Rangkorrelationen nach Spearman r_s gebildet werden müssen, vgl. [Bor99, 214].

Tabelle 8.4: Signifikante Korrelationen.

Abhängige Variable	Kontrollvariable	Treatmentfaktor
		α-Fehler
W-Index	S13	Lehrmethoden
		4.1 %
	S15	Lehrmethoden
		4.7 %

Tabelle 8.5: Signifikantes Ergebnis zweier Kovarianzanalysen mit den Kontrollvariablen S13 und S15.



(a) S13 (b) S15

Abbildung 8.4: Geschätzte Leistungsdifferenzen zwischen den Lehrmethoden für den W-Index unter Einbeziehung der Kontrollvariablen S13 und S15.

weiteren Unterschiede zwischen den Gruppen. Dies war auch für die Variable S7 festzustellen, die mit dem W-Index stark korreliert. Anzumerken bleibt dabei, daß die Gruppen im Durchschnitt gute Mathematiknoten hatten (Notendurchschnitt von 1.79–1.89). Die mittleren Leistungen in den Wissensfragen waren sehr gut (5.02–6.00, vgl. Tabelle 8.2), da ein Höchstwert von sieben Punkten zu erreichen war. Dies mag einerseits an den hohen Durchschnittswerten in den Mathematiknoten liegen, andererseits daran, daß sich bereits 89.7 % der Versuchspersonen mit dem Lerngebiet beschäftigt hatten. Betrachtet man nur diejenigen Versuchspersonen, die sich im Studium schon mit dem Lerngebiet befaßt haben, zeigen sich ebenfalls keine Unterschiede zwischen den Gruppen. Es wäre möglich, daß die Wissensfragen zu einfach waren und daher Unterschiede in den Lernerfolgen durch die verschiedenen Methoden verdeckt wurden. Differenzen bzgl. der *Transferfragen* zeigten sich zwischen den Gruppen auch unter Einbeziehung möglicher weiterer Einflußfaktoren nicht.

8.4.3 Ergebnis der Lehrmittelbewertung

Dieser Abschnitt vergleicht die untersuchten Gruppen bzgl. der in Anhang B.2 angegebenen 12 Bewertungsfragen durch jeweils eine Varianzanalyse. Die Bewertungsfragen bezogen sich auf die Medien, mit denen das Teilgebiet der lexikalischen Analyse gelernt worden war: fünf Fragen (B6–B10) wurden nur den Versuchspersonen der beiden Softwaregruppen und Frage B11 nicht der Vorlesungsgruppe gestellt. Die Fragen setzten sich aus 11-stufigen Rating-Skalen und numerischen Marken sowie aus „offenen“ Fragen zusammen, zu denen die Versuchspersonen ihre Antworten schriftlich ausformulieren konnten.

Auswertung der Rating-Fragen

Die Auswertungen der Rating-Fragen werden durch Histogramme dargestellt, welche jeweils den Skalenmittelwert der entsprechenden Gruppe und die dazugehörigen Standardabweichungen (durch Fehlerbalken symbolisiert) verdeutlichen. Signifikante Unterschiede sind durch Sternchen *, sehr signifikante Differenzen durch ** neben der entsprechenden Variable gekennzeichnet und werden explizit erläutert.

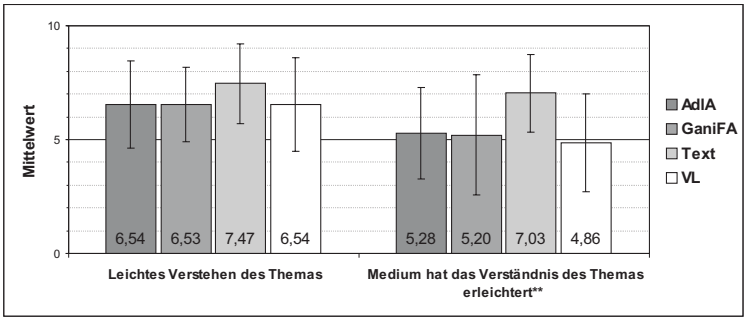


Abbildung 8.5: Resultate der Bewertungsfragen B1 und B2.

Die Fragen B1 und B2 (vgl. Abb. 8.5) beleuchten die subjektiv empfundene Komplexität des Lerngebiets und die Auswirkungen der vier Lehrmethoden auf diese Komplexität. Die Differenzen der Gruppen kamen bei Frage B2 nur noch zu 0.1 % zufällig zustande und sind daher sehr signifikant. Hier schnitt der Text als Lernmittel am besten ab.

Bei der Frage B5 (vgl. Abb. 8.6) sollte das spezielle Medium, mit dem gelernt wurde, mit dem Lehrbuch „Hardware Design“ von Keller und Paul [KP97] verglichen werden. Nur 7.8 % der VP kannten das Lehrbuch nicht, da es in dem entsprechenden Untersuchungszeitraum als Standardwerk eingesetzt wurde. Im Vergleich hinsichtlich der Teilfrage, ob das Lernen mit dem jeweiligen Medium sehr viel weniger anstrengend war, wurden die Lehrmethoden signifikant unterschiedlich beurteilt. Auch hier schnitt der Text als Lernhilfe wieder am besten ab. Betrachtet man die einzelnen Skalenmittelwerte der verschiedenen Gruppen, fällt auf, daß alle vier Lehrmethoden im Vergleich zum o. g. Lehrbuch besser beurteilt wurden (Mittelwerte über fünf). Das Lehrbuch scheint nicht sehr beliebt bei den Studenten zu sein. Dies könnte weitere Unterschiede bei dieser Frage zwischen den Gruppen überdeckt haben.

Frage B6 (vgl. Abb. 8.7) läßt auf die Akzeptanz der Lernsoftware schließen. Sowohl das Lernsystem AdlA als auch GaniFA würden vermutlich gerne als Ergänzung zu traditionellen Lehrmitteln angewendet werden. Die Fragen B8 und B10 untersuchen die Benutzerfreundlichkeit beider Systeme hinsichtlich der Animationen. Allgemein wurde diese Eigenschaft als sehr gut beurteilt, im Fall der Frage B8 zur Animationssteuerung ist der resultierende Unterschied zwischen beiden Systemen sogar signifikant.

Zur Bewertung der didaktischen Aufbereitung wurden die Fragen B11 und B12 (vgl. Abb. 8.8) gestellt. Die entsprechenden Varianzanalysen zeigten keinen signifikanten Unterschied zwischen den Gruppen, obwohl die erreichten Stichprobenmittelwerte für die Softwaregruppen zu Frage B11 ausgezeichnet waren.

Auswertung der offenen Fragen

Offene Fragen geben genauere Auskünfte darüber, wie man eine Lernmethode verbessern kann. Es wurden vier offene Fragen (B3, B4, B7, B9) gestellt, die jeweils positive/negative Aspekte und Anregungen zur Verbesserung der Lernmethode erfaßten. Klassifikationen der Antworten und ihrer Häufigkeiten sind in Anhang B.2 unter der entsprechenden Frage aufgelistet.

Allgemein wurden die generierten Animationen des elektronischen Textbuchs GaniFA als gut beurteilt. Dies gilt ebenso für die Graphiken, die didaktische Aufbereitung und die formalen Definitionen in einem separaten Definitionsfenster. Letztere wurden allerdings als „zu formal“ angesehen. Probleme schien die nicht intuitive Eingabe der regulären Ausdrücke zu bereiten. Auf Mißfallen stieß bei mehreren Versuchspersonen die graphische Aufmachung der HTML-Seiten und eine nicht ausreichende Erklärung der Farbsymbolik in den Animationen. In Übereinstimmung mit dem o. g. Resultat zur Rating-Frage B6 würden viele Versuchspersonen GaniFA aufgrund der Animationen, Übersichtlichkeit und besseren Motivation als Ergänzung zu einer Vorlesung verwenden. Andererseits gaben einige Probanden an, daß sie grundsätzlich lieber aus Büchern lernen.

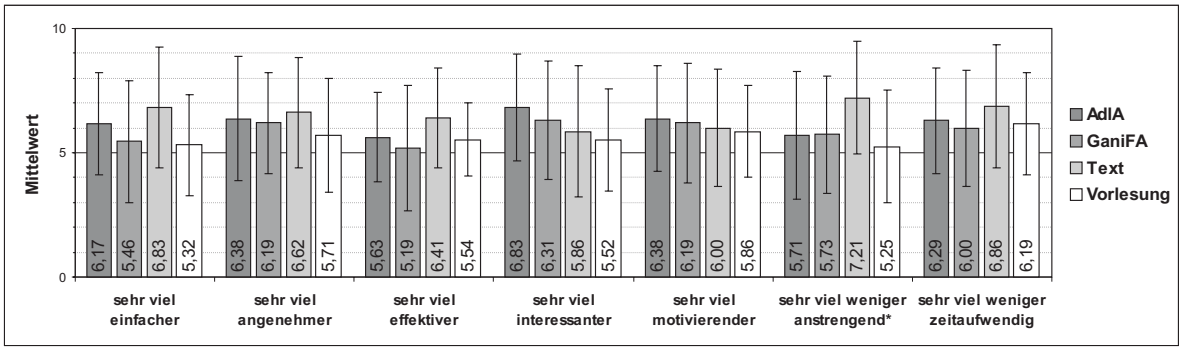


Abbildung 8.6: Resultat der Bewertungsfrage B5.

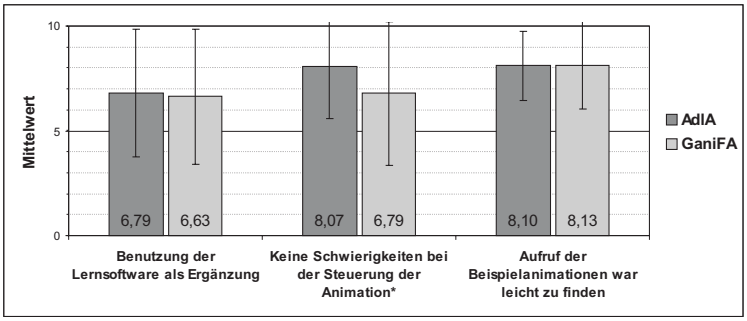
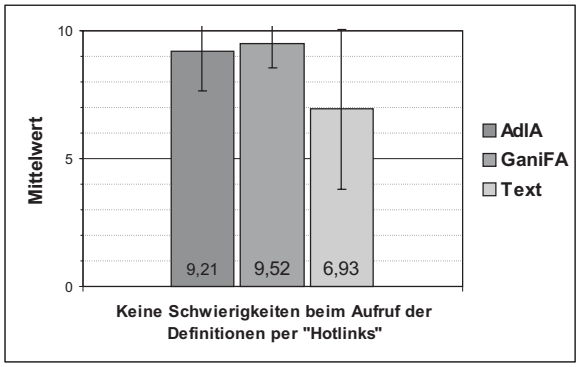
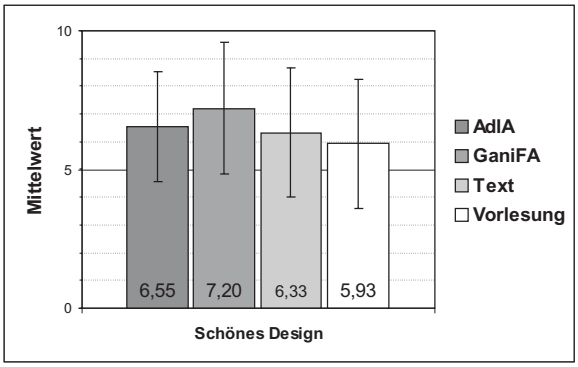


Abbildung 8.7: Resultate der Bewertungsfragen B6, B8 und B10.



(a) B11



(b) B12

Abbildung 8.8: Resultate der Bewertungsfragen B11 und B12.

8.4.4 Zugriffstatistik des Textbuchs

Die Zugriffe auf das elektronische Textbuch GaniFA über das WWW wurden anhand eines Logfiles des Webservers analysiert. Hierbei wurde ein Zeitraum von 27 Tagen (vom 17. Mai bis 13. Juni 2001) ausgewertet. In dieser Zeitspanne wurde insgesamt von 479 IP-Adressen (keine Suchmaschinen) auf mindestens eine Webseite des Textbuchs zugegriffen, wobei alle Zugriffe von derselben IP-Adresse, die nicht innerhalb einer Stunde geschahen, mehrfach gezählt wurden. 215 hiervon waren Zugriffe auf mindestens drei Webseiten des Textbuchs. Dies entspricht durchschnittlich acht Benutzerzugriffen pro Tag auf das Textbuch.

Die Möglichkeit der Algorithmenanimation durch das eingebettete GaniFA-Applet wurde von vielen Benutzern angenommen. Über die Hälfte der Benutzer, die mindestens drei Webseiten des Textbuchs besuchten, griffen auch auf mindestens eine Webseite mit dem eingebetteten GaniFA-Applet zu. Wieviele dieser Benutzer in welchem Umfang das Applet auch wirklich ausführten, konnte mit unserer Analyse nicht berechnet werden.

8.4.5 Zusammenfassung

Eine Evaluation überprüft die Leistungsfähigkeit und Benutzerfreundlichkeit eines Lernsystems. GaniFA wurde mit drei weiteren Lernmethoden verglichen und eine deskriptive Analyse ergab für GaniFA den höchsten Leistungswert bei der Beantwortung von Transferfragen. Eine anschließende Varianzanalyse zeigte allerdings keine Unterschiede im Lernerfolg der Versuchspersonen. Damit war im Umkehrschluß die Nullhypothese zunächst signifikant belegt: es existierte kein relevanter Leistungsunterschied zwischen den Gruppen.

Die Einbeziehung zweier Störvariablen ergab für die Störvariable S15 (Informatikkurs) eine signifikante Leistungsdifferenz bei den Wissensfragen zugunsten der Vergleichsgruppe, die mit einem äquivalenten Lehrtext gearbeitet hatte. Die GaniFA-Gruppe lag unter Kontrolle der Störvariablen S13 (Mathematikkurs) mit der Textgruppe gleichauf und erzielte unter Kontrolle beider Störvariablen einen Leistungszuwachs vor der AdlA- und der Vorlesungsgruppe. Daraus ergibt sich für das GaniFA-System und den in Abschnitt 7.4 diskutierten generativen Ansatz zweiter Ordnung eine solide Basis für weitere Entwicklungen. Zukünftige Lernexperimente sollten allerdings einen Vortest zur Überprüfung des Vorwissens der VP mit einbeziehen, da die Wissensfragen evtl. zu einfach gewählt und daher Unterschiede in den Lernerfolgen verdeckt worden sind.

Gerade die offenen Nennungen zeigten, daß sich GaniFA als Ergänzung zu üblichen Lernmethoden (etwa zu einer Vorlesung) gut eignet. Das System bietet hier Vorteile, die keine der anderen Lernmethoden besitzt: Animationen, die selbst generiert werden können. Die Versuchspersonen schätzten es sehr, eigenständig zu arbeiten und die Möglichkeiten dieser Animationen auszuschöpfen. Besondere Anerkennung wurde der guten Übersichtlichkeit der präsentierten Lerninhalte sowie der motivierenden Eigenschaften des Systems zuteil. GaniFA wurde als Lernmethode sehr positiv wahrgenommen, und die meisten Versuchspersonen würden GaniFA als Ergänzung zur Vorlesung gerne benutzen.

9 Abschließende Bemerkungen

In dieser Arbeit haben wir das GANIMAL-Framework vorgestellt, das für die Entwicklung interaktiver, multimedialer Visualisierungen und Animationen von verschiedenen Übersetzerphasen verwendet werden kann. Diese Visualisierungen und Animationen können in eine web-basierte Lehr- und Lernsoftware eingebettet werden. In diesem abschließenden Kapitel fassen wir unsere Arbeit zusammen und geben einen Ausblick auf zukünftige Forschungsfelder, insbesondere auf mögliche Erweiterungen des Frameworks und zusätzliche Anwendungen.

9.1 Zusammenfassung

Das GANIMAL-Framework vereinigt mehrere klassische Konzepte aus verschiedenen Systemen zur Entwicklung von Algorithmenanimationen, z. B. Interesting Events, Sichten oder parallele Ausführung von Programmpunkten. Wir haben im ersten Teil dieser Arbeit eine Reihe neuerer Systeme kurz diskutiert, klassifiziert und die dort realisierten Konzepte bzw. Ansätze erläutert. Die zu Beginn der Entwicklungsphase von GANIMAL vorhandenen Algorithmenanimationssysteme waren meist nicht in JAVA implementiert, zum Großteil plattformabhängig oder nur für einfache Animationen von Sortierverfahren geeignet. Die mit ihnen erstellten Animationen entsprachen oft nicht modernen, plattformunabhängigen Standardtechnologien wie etwa JAVA-Applets und konnten damit nicht mit jedem Standardwebbrowser betrachtet werden. Erst Ende der 90er Jahre und zu Beginn des neuen Jahrtausends wurden neue Systeme vorgestellt, die diese Nachteile beheben konnten. Zu diesen Systemen ist auch GANIMAL zu zählen, das vollständig in JAVA implementiert wurde.

Unsere Animationsbeschreibungssprache GANILA bietet eine Fülle neuer Möglichkeiten, die über die Eigenschaften anderer bekannter Systeme hinausgehen: alternative Interesting Events, Vermischung von post mortem- und live/online-Algorithmenanimationen, Vorausschauendes Layout von Graphen, Animationssteuerung der Besuchsfolge von Schleifen und Testen von Invarianten während der Ausführung von Programmpunktgruppen. GANILA ist eine Erweiterung von JAVA und wird von einem Compiler in reines JAVA übersetzt. Dazu wurden die einzelnen Programmpunkte des Eingabeprogramms reifiziert. Die dadurch mögliche Verknüpfung von Metainformationen mit einzelnen Programmpunkten ist unseres Wissens neu in der Softwarevisualisierung. Der übersetzte Programmcode bildet in Kombination mit der GANIMAL-Laufzeitumgebung interaktive Animationen des Eingabeprogramms. Die Laufzeitumgebung unterstützt den Entwickler einer Animation u. a. in der Implementierung eigener Sichten auf den Eingabealgo-

rithmus und bietet eine Reihe von vordefinierten Standardsichten. Es übernimmt die während der Parallelausführung von Programmpunkten notwendigen Synchronisationsaufgaben, unterstützt ein variables Zeitmanagement von Interesting Events und enthält die Implementierung einer graphischen Benutzeroberfläche zur Animationssteuerung sowie zur Veränderung der o. g. Metainformationen.

Die Anwendbarkeit unseres Systems wurde anhand einer Implementierung der Animation des Heapsort-Algorithmus verdeutlicht. Eine komplexere Anwendung des Frameworks ist das GaniFA-Applet, eine Generatorimplementierung für die Erzeugung endlicher Automaten aus regulären Ausdrücken. Das Applet wurde in ein elektronisches Textbuch über die Generierung und Ausführung der lexikalischen Analyse eingebettet, das von der Homepage des Ganimal-Projekts [Gan02b] heruntergeladen und getestet werden kann.

Wir diskutierten grundlegende lerntheoretische Aspekte und stellten ein Lernmodell vor, das sowohl unsere älteren Lernsysteme, wie beispielsweise die Systeme AdsA oder Ganimam, als auch die Realisierung des generativen Ansatzes von GaniFA in vier unterschiedliche Stufen explorativen Lernens einordnet. Es wurden auch pädagogische Vor- und Nachteile der Systeme hervorgehoben. In einem Lernexperiment mit über 100 Studenten wurde jeweils eine Lernsoftware über endliche Automaten ohne und mit generativem Teil evaluiert. Die Ergebnisse zeigen, daß unter Einbeziehung zweier Störvariablen beide Lernsysteme im Vergleich zu einer konventionellen Vorlesung mit identischem Lernstoff besser, aber zu einem Lehrbuch gleich gut bis etwas schlechter abschneiden. Die in dieser Arbeit präsentierte statistische Analyse läßt darauf schließen, daß ein Teil der in dieser Evaluation gestellten Fragen möglicherweise zu einfach gewählt wurde. Zukünftige Evaluationen sollten daher im Vorfeld des Lernexperiments einen Test durchführen, um das Vorwissen der Probanden beurteilen und angemessen berücksichtigen zu können. Darüber hinaus empfanden die Versuchspersonen die generativen Möglichkeiten des GaniFA-Lernsystems, d. h. die mögliche Eingabe regulärer Ausdrücke und die Beobachtung des Generierungsprozesses als in hohem Maße motivationssteigernd. Die meisten von ihnen würden GaniFA als Zusatz zu einer Vorlesung oder zu einem Lehrbuch sehr gerne verwenden.

9.2 Ausblick

Das Ganimal-Framework eignet sich natürlich nicht nur für die Implementierung von lexikalischen Analysegeneratoren und den daraus resultierenden Animationen des Akzeptanzverhaltens endlicher Automaten sowie ihres Generierungsprozesses. Es lassen sich auch komplexere Generierungsprozesse im Übersetzerbau, z. B. die Generierung von Kellerautomaten oder Attributauswertern, mit Hilfe unseres Systems visualisieren. Insbesondere Ganila und das Laufzeitsystem mit seinem graphischen Basispaket könnten in späteren Projekten auch zur Entwicklung von Generatoren für andere prozeßorientierte Lerninhalte wie Algorithmen, Rechnerarchitektur, chemische Reaktionen oder Modellierung ökonomischer Prozesse eingesetzt werden. Um die Anwendbarkeit des Frameworks noch zu steigern, könnte es um die nachfolgenden Eigenschaften erweitert werden:

- Im gegenwärtigen Entwicklungszustand von GANIMAL können Benutzereingaben zur Laufzeit der Algorithmenanimation nur sehr umständlich über das Konsolenfenster vorgenommen werden. Ein neue Standardsicht IoView könnte hier Abhilfe schaffen. Sie würde auf der in dieser Arbeit beschriebenen HtmlView basieren und dem Animationsentwickler ermöglichen, Eingabefelder in HTML zu spezifizieren. Ein Anwender könnte dann über dieses Feld eigene Werte eingeben oder über ein Auswahlfeld vorgegebene Werte selektieren. Um die Werte zurückzuliefern, könnte man einerseits den entsprechenden Interesting Events Objektreferenzen als Argument übergeben. Andererseits könnte man die Sprache GANILA um eine besondere Art von Interesting Events mit Rückgabewerten erweitern. Die bisherige Architektur des Frameworks sieht diese Möglichkeit aber nicht vor.

- Eine Animationssteuerung von Schleifen wurde in das GANIMAL-Framework integriert. Analog könnte man eine Animationssteuerung von Rekursionen implementieren, also z. B. die Visualisierung der ersten drei rekursiven Aufrufe einer Methode.

- Von ebenfalls großem Nutzen wäre eine zusätzliche Kommunikationssicht: Informationen, die zur Laufzeit an bestimmten Programmpunkten zur Verfügung stehen, könnten durch spezielle Interesting Events an CGI-Skripten zu einem Server transferiert und dort weiterverarbeitet werden. Verschiedene Anwender könnten beispielsweise über einen Server koordiniert werden etc.

- Pädagogisch sinnvoll wäre eine Erweiterung der Invariantensicht, in der Lernende zur Laufzeit der Animation selbst Hypothesen formulieren und testen können. Der Animationsentwickler könnte diesen Prozeß durch Vorgabe von interaktiven Methoden unterstützen.

- Schließlich könnte eine zusätzliche Variablensicht alle aktuellen Werte globaler und lokaler Variablen graphisch anzeigen. Eine solche Sicht würde vom aktuellen Zustand des zu animierenden Algorithmus abhängen (zustandsgetriebene Animation), d. h. Änderungen würden nicht — wie in unserem System üblich — von Events spezifiziert.

A Quellcode der Animation von Heapsort

In Abschnitt 6.2 wurde aufgezeigt, wie eine Algorithmenanimation unter Verwendung des GANIMAL-Frameworks erstellt werden kann. In diesem Anhang sind die vollständigen Quellcodes für die Implementierung einer Animation des Heapsort-Algorithmus abgedruckt. Teil A.1 enthält den GANILA-Programmcode, das generierte Algorithmenmodul sowie einen Ausschnitt der generierten initialen Programmpunkteinstellungen. Teil A.2 zeigt die Implementierung der in Abschnitt 6.2 diskutierten Balkensicht. Die dort ebenfalls beschriebenen Baum- und Arraysichten sind in diesem Anhang weggelassen worden, da sie ähnlich implementiert werden und keinen neuen Erkenntnisgewinn bringen. Der letzte Teil dieses Anhangs, A.3, beinhaltet die Implementierung einer JAVA-Klasse, in der die verschiedenen Sichten in einem Fenster zusammengefaßt und im Laufzeitsystem registriert werden.

A.1 Implementierung des Heapsort-Algorithmus

A.1.1 Spezifikation in GANILA

Die nachfolgend angegebene GANILA-Spezifikation enthält an PP 32 ein zusätzliches Interesting Event, das eine Audiodatei über die SoundView abspielen läßt. Diese wird im Vorspann der GANILA-Klasse entsprechend deklariert.

```
package ganimal.applications.heapsort;

view SoundView(new String[] {"break.wav"});

ganila class Heapsort {
  int A[] = new int[]{ 32, 8, 1, 12, 32, 26, 2, 9, 10, 5 };
  int heapSize;

  /**
   * Aufbau des initialen Heaps
   * Schleife für das Entfernen der Wurzel und
   * Wiederherstellen der Heapeigenschaft
   */
  public void start(String[] argv) {
    heapSize = A.length - 1;                                    // PP 0
```

```
    buildheap();                                                    // PP 1

    for (int i = A.length - 1; i >= 0; i--) {                       // PP 2
      exchange(0, i);                                               // PP 3
      *IE_DisableNode(i);                                           // PP 4
      heapSize--;                                                   // PP 5
      Heapify(0);                                                   // PP 6
    }
  }

  /**
   * Stellt die Heapeigenschaft für einen Teilbaum mit Wurzel i her
   */
  public void Heapify(int i) {
    int l, r, largest;

    l = Left(i);                                                    // PP 7
    r = Right(i);                                                   // PP 8

    if (l <= heapSize) {                                            // PP 9
      *IE_MarkNodes(i, l);                                          // PP 10
      *IE_CompareNodes(i, l);                                       // PP 11
    }

    if (l <= heapSize && A[l] > A[i]) {                             // PP 12
      *IE_MarkMax(l, i);                                            // PP 13
      largest = l;                                                  // PP 14
    } else {
      if (l <= heapSize)                                            // PP 15
        *IE_MarkMax(i, l);                                          // PP 16
      largest = i;                                                  // PP 17
    }

    if (r <= heapSize) {                                            // PP 18
      *IE_MarkNodes(largest, r);                                    // PP 19
      *IE_CompareNodes(largest, r);                                 // PP 20
    }

    if (r <= heapSize && A[r] > A[largest]) {                       // PP 21
      *IE_MarkMax(r, largest);                                      // PP 22
      largest = r;                                                  // PP 23
    } else {
      if (r <= heapSize)                                            // PP 24
        *IE_MarkMax(largest, r);                                    // PP 25
    }
```

```
  if (largest != i) {                                          // PP 26
    exchange(i, largest);                                      // PP 27
    Heapify(largest);                                          // PP 28
  }
}

/**
 * Vertauscht die Elemente mit Index i und j eines Feldes
 */
public void exchange(int i, int j)  {
  int help;

  *{ *IE_MoveToTemporary(i, A);                                // PP 30
     help = A[i];                                              // PP 31
     *BREAK;
     *IE_PlaySound("break.wav");                               // PP 32
     *IE_MoveElement(i, j, A);                                 // PP 33
     A[i] = A[j];                                              // PP 34
     *IE_MoveFromTemporary(j, A);                              // PP 35
     A[j] = help;                                              // PP 36
  *} *FOLD *{                                                  // PP 29
     *IE_Exchange(i, j, A);                                    // PP 37
  *}
}

/**
 * Berechnet den initialen Heap
 * Schleifenaufruf von Heapify
 */
public void buildheap() {
  heapSize = A.length - 1;                                     // PP 38
  for (int i = (int)Math.floor(A.length / 2); i >= 0; i--) {   // PP 39
    Heapify(i);                                                // PP 40
  }
}

/**
 * Liefert Index des linken Kindes
 */
public int Left(int i) {
  return 2 * i + 1;                                            // PP 41
}

/**
 * Liefert Index des rechten Kindes
 */
```

```
  public int Right(int i) {
    return 2 * i + 2;                                                    // PP 42
  }
}
```

A.1.2 Generierte Module

Algorithmenmodul

```
package ganimal.applications.heapsort;

import ganimal.base.GView;
import ganimal.runtime.*;
import ganimal.runtime.event.*;
import ganimal.runtime.views.aura.*;
import ganimal.runtime.type.*;

public class HeapsortAlgorithm extends GAlgorithm {
   Gint A[] = new Gint[] {
      new Gint(32), new Gint(8), new Gint(1), new Gint(12),
      new Gint(32), new Gint(26), new Gint(2), new Gint(9),
      new Gint(10), new Gint(5)
   };
   Gint heapSize = new Gint();

   public HeapsortAlgorithm() {
      super(new Heapsort_AST());
      StandardViews = new GView[] {
         new SoundView(new String[]{ "break.wav" })
      };
      addViewsToControl();
   }

   public void start(String[] argv) {
      dispatch(0, null);
      dispatch(1, null);
      dispatch(2, null);
   }

   public void Heapify(Gint i) {
      Gint l = new Gint(), r = new Gint(), largest = new Gint();
      dispatch(7, new Object[]{ i, l });
      dispatch(8, new Object[]{ i, r });
      dispatch(9, new Object[]{ i, l });
      dispatch(12, new Object[]{ i, l, largest });
      dispatch(18, new Object[]{ largest, r });
      dispatch(21, new Object[]{ largest, r });
```

```
   dispatch(26, new Object[]{ i, largest });
}

public void exchange(Gint i, Gint j) {
   Gint help = new Gint();
   dispatch(29, new Object[]{ help, i, j });
}

public void buildheap() {
   dispatch(38, null);
   dispatch(39, null);
}

public Gint Left(Gint i) {
   ControlFlow cfl;
   cfl = dispatch(41, new Object[]{ i });
   return (Gint) cfl.value;
}

public Gint Right(Gint i) {
   ControlFlow cfl;
   cfl = dispatch(42, new Object[]{ i });
   return (Gint) cfl.value;
}

public void dispatch_0() { heapSize.setValue(A.length - 1); }

public void dispatch_1() { buildheap(); }

public void dispatch_2() {
   for (Gint i = new Gint(A.length - 1); i.getValue() >= 0; i.postDec()) {
      dispatch(3, new Object[]{ i });
      dispatch(4, new Object[]{ i });
      dispatch(5, null);
      dispatch(6, null);
   }
}

public void dispatch_3(Gint a1) { exchange(new Gint(0), a1.Clone()); }

public void dispatch_4(Gint a1) {
   GEvent e;
   e = new GEvent(4, isRecord(), isVisibleEvent(), getLoReDepth()
      "DisableNode", new Object[]{ a1 });
   control.broadcast(e);
}
```

```
public void dispatch_5() { heapSize.postDec(); }

public void dispatch_6() { Heapify(new Gint(0)); }

public void dispatch_7(Gint a1, Gint a2) {
   a2.setValue(Left(a1.Clone()).getValue());
}

public void dispatch_8(Gint a1, Gint a2) {
   a2.setValue(Right(a1.Clone()).getValue());
}

public void dispatch_9(Gint a1, Gint a2) {
   if (a2.getValue() <= heapSize.getValue()) {
      dispatch(10, new Object[]{ a1, a2 });
      dispatch(11, new Object[]{ a1, a2 });
   }
}

public void dispatch_10(Gint a1, Gint a2) {
   GEvent e;
   e = new GEvent(10, isRecord(), isVisibleEvent(), getLoReDepth(),
      "MarkNodes", new Object[]{ a1, a2 });
   control.broadcast(e);
}

public void dispatch_11(Gint a1, Gint a2) {
   GEvent e;
   e = new GEvent(11, isRecord(), isVisibleEvent(), getLoReDepth(),
      "CompareNodes", new Object[]{ a1, a2 });
   control.broadcast(e);
}

public void dispatch_12(Gint a1, Gint a2, Gint a3) {
   if (a2.getValue() <= heapSize.getValue() &&
         A[a2.getValue()].getValue() > A[a1.getValue()].getValue()) {
      dispatch(13, new Object[]{ a1, a2 });
      dispatch(14, new Object[]{ a2, a3 });
   } else {
      dispatch(15, new Object[]{ a1, a2 });
      dispatch(17, new Object[]{ a1, a3 });
   }
}

public void dispatch_13(Gint a1, Gint a2) {
```

```
   GEvent e;
   e = new GEvent(13, isRecord(), isVisibleEvent(), getLoReDepth(),
      "MarkMax", new Object[]{ a2, a1 });
   control.broadcast(e);
}

public void dispatch_14(Gint a1, Gint a2) { a2.setValue(a1.getValue()); }

public void dispatch_15(Gint a1, Gint a2) {
   if (a2.getValue() <= heapSize.getValue()) {
      dispatch(16, new Object[]{ a1, a2 });
   }
}

public void dispatch_16(Gint a1, Gint a2) {
   GEvent e;
   e = new GEvent(16, isRecord(), isVisibleEvent(), getLoReDepth(),
      "MarkMax", new Object[]{ a1, a2 });
   control.broadcast(e);
}

public void dispatch_17(Gint a1, Gint a2) { a2.setValue(a1.getValue()); }

public void dispatch_18(Gint a1, Gint a2) {
   if (a2.getValue() <= heapSize.getValue()) {
      dispatch(19, new Object[]{ a1, a2 });
      dispatch(20, new Object[]{ a1, a2 });
   }
}

public void dispatch_19(Gint a1, Gint a2) {
   GEvent e;
   e = new GEvent(19, isRecord(), isVisibleEvent(), getLoReDepth(),
      "MarkNodes", new Object[]{ a1, a2 });
   control.broadcast(e);
}

public void dispatch_20(Gint a1, Gint a2) {
   GEvent e;
   e = new GEvent(20, isRecord(), isVisibleEvent(), getLoReDepth(),
      "CompareNodes", new Object[]{ a1, a2 });
   control.broadcast(e);
}

public void dispatch_21(Gint a1, Gint a2) {
   if (a2.getValue() <= heapSize.getValue() &&
```

```
            A[a2.getValue()].getValue() > A[a1.getValue()].getValue()) {
         dispatch(22, new Object[]{ a1, a2 });
         dispatch(23, new Object[]{ a1, a2 });
      } else {
         dispatch(24, new Object[]{ a1, a2 });
      }
   }

   public void dispatch_22(Gint a1, Gint a2) {
      GEvent e;
      e = new GEvent(22, isRecord(), isVisibleEvent(), getLoReDepth(),
         "MarkMax", new Object[]{ a2, a1 });
      control.broadcast(e);
   }

   public void dispatch_23(Gint a1, Gint a2) { a1.setValue(a2.getValue()); }

   public void dispatch_24(Gint a1, Gint a2) {
      if (a2.getValue() <= heapSize.getValue()) {
         dispatch(25, new Object[]{ a1, a2 });
      }
   }

   public void dispatch_25(Gint a1, Gint a2) {
      GEvent e;
      e = new GEvent(25, isRecord(), isVisibleEvent(), getLoReDepth(),
         "MarkMax", new Object[]{ a1, a2 });
      control.broadcast(e);
   }

   public void dispatch_26(Gint a1, Gint a2) {
      if (a2.getValue() != a1.getValue()) {
         dispatch(27, new Object[]{ a1, a2 });
         dispatch(28, new Object[]{ a2 });
      }
   }

   public void dispatch_27(Gint a1, Gint a2) {
      exchange(a1.Clone(), a2.Clone());
   }

   public void dispatch_28(Gint a1) { Heapify(a1.Clone()); }

   public ControlFlow dispatch_29(Gint a1, Gint a2, Gint a3) {
      return executeFolding(29, new Object[]{ a1, a2, a3 });
   }
```

```
public void dispatch_30(Gint a1) {
   GEvent e;
   e = new GEvent(30, isRecord(), isVisibleEvent(), getLoReDepth(),
      "MoveToTemporary", new Object[]{ a1, A });
   control.broadcast(e);
}

public void dispatch_31(Gint a1, Gint a2) {
   a1.setValue(A[a2.getValue()].getValue());
}

public void dispatch_32() {
   GEvent e;
   e = new GEvent(32, isRecord(), isVisibleEvent(), getLoReDepth(),
      "PlaySound", new Object[]{ "break.wav" });
   control.broadcast(e);
}

public void dispatch_33(Gint a1, Gint a2) {
   GEvent e;
   e = new GEvent(33, isRecord(), isVisibleEvent(), getLoReDepth(),
      "MoveElement", new Object[]{ a1, a2, A });
   control.broadcast(e);
}

public void dispatch_34(Gint a1, Gint a2) {
   A[a1.getValue()].setValue(A[a2.getValue()].getValue());
}

public void dispatch_35(Gint a1) {
   GEvent e;
   e = new GEvent(35, isRecord(), isVisibleEvent(), getLoReDepth(),
      "MoveFromTemporary", new Object[]{ a1, A });
   control.broadcast(e);
}

public void dispatch_36(Gint a1, Gint a2) {
   A[a2.getValue()].setValue(a1.getValue());
}

public void dispatch_37(Gint a1, Gint a2) {
   GEvent e;
   e = new GEvent(37, isRecord(), isVisibleEvent(), getLoReDepth(),
      "Exchange", new Object[]{ a1, a2, A });
   control.broadcast(e);
```

```
   }

   public void dispatch_38() { heapSize.setValue(A.length - 1); }

   public void dispatch_39() {
      for (Gint i = new Gint((int)Math.floor(A.length / 2));
            i.getValue() >= 0; i.postDec()) {
         dispatch(40, new Object[]{ i });
      }
   }

   public void dispatch_40(Gint a1) { Heapify(a1.Clone()); }

   public ControlFlow dispatch_41(Gint a1) {
      return new ControlFlow("return", new Gint(2 * a1.getValue() + 1));
   }

   public ControlFlow dispatch_42(Gint a1) {
      return new ControlFlow("return", new Gint(2 * a1.getValue() + 2));
   }

   public ControlFlow execute_29_1(Gint a1, Gint a2, Gint a3) {
      dispatch(30, new Object[]{ a2 });
      dispatch(31, new Object[]{ a1, a2 });
      dispatch(32, null);
      dispatch(33, new Object[]{ a2, a3 });
      dispatch(34, new Object[]{ a2, a3 });
      dispatch(35, new Object[]{ a3 });
      dispatch(36, new Object[]{ a1, a3 });
      return new ControlFlow();
   }

   public ControlFlow execute_29_2(Gint a1, Gint a2, Gint a3) {
      dispatch(37, new Object[]{ a2, a3 });
      return new ControlFlow();
   }
}
```

Initiale Programmpunkteinstellungen

GaJa erzeugt für jeden Programmpunkt initiale PPEs. Diese sind für die Programmpunkte 29-37 der Methode `exchange()` im folgenden angegeben:

```
st[29] = new Settings(29, new Object[]{new Folding_Setting(true)});
st[30] = new Settings(30, new Object[]{new IE_Setting(true)});
st[32] = new Settings(32, new Object[]{new Breakpoint_Setting(true),
                                       new IE_Setting(true)});
```

```
st[33] = new Settings(33, new Object[]{new IE_Setting(true)});
st[35] = new Settings(35, new Object[]{new IE_Setting(true)});
st[37] = new Settings(37, new Object[]{new IE_Setting(true)});
```

Alle anderen PPEs haben für den hier spezifizierten Algorithmus einen leeren Eintrag, d. h. keine zusätzlichen Metainformationen, die zur Laufzeit geändert werden könnten.

A.2 Implementierung der Balkensicht

Da die Eventhandler der im Algorithmus spezifierten Events `*IE_MoveToTemporary`, `*IE_MoveElement` und `*IE_MoveFromTemporary` die Darstellung des Quellcodes zu unübersichtlich machen würden und diese Events für eine einfache Animation des Heapsort-Algorithmus nicht unbedingt notwendig sind, wurde hier auf deren Abdruck verzichtet.

```
package ganimal.applications.heapsort;

import java.awt.*;
import ganimal.base.*;
import ganimal.runtime.type.*;

public class BarView extends HeapsortView {
  GBarRenderer barRenderer;

  public BarView(Gint[] array) {
    setName("Bar View");
    barRenderer = new GBarRenderer(array, GBarRenderer.BAR3D_MODE, true);
    addRenderer(barRenderer);
  }

  /**
   * Interesting Event : MarkNodes
   * Markiere nacheinander die Balken der Indizes i und j in gelber Farbe.
   */
  public void IE_MarkNodes_Play_Visible(Gint i, Gint j, Gint[] A) {
    barRenderer.setBackgroundColor(Color.yellow, i.getValue());
    waitms(1000);
    barRenderer.setBackgroundColor(Color.yellow, j.getValue());
    waitms(2000);
  }

  /**
   * Interesting Event : MarkMax
   * Markiere zunächst den Balken des Maximums A[i] in roter und den des
   * Wertes A[j] in hellgrauer Farbe. Färbe dann beide Balken hellgrau.
   */
  public void IE_MarkMax_Play_Visible(Gint i, Gint j, Gint[] A) {
```

```
    barRenderer.setBackgroundColor(Color.red, i.getValue());
    barRenderer.setBackgroundColor(Color.lightGray, j.getValue());
    waitms(2000);
    barRenderer.setBackgroundColor(Color.lightGray, i.getValue());
  }

  /**
   * Interesting Event : DisableNode
   * A[i] ist der maximale Wert an der Wurzel des Heaps und wird entfernt.
   * In der Balkensicht wird der Balken mit dem Index i grün gefärbt.
   */
  public void IE_DisableNode_Play_Visible(Gint i, Gint[] A) {
    barRenderer.setBackgroundColor(Color.green, i.getValue());
    waitms(1000);
  }

  /**
   * Interesting Event : Exchange
   * Zeichne die Sicht neu. Die Balken werden automatisch vertauscht.
   */
  public void IE_Exchange_Play_Visible(Gint i, Gint j, Gint[] A) {
    barRenderer.invalidate();
  }
}
```

A.3 Integration der graphischen Komponenten

```
package ganimal.applications.heapsort;

import ganimal.base.*;
import ganimal.runtime.event.*;
import ganimal.runtime.*;
import java.awt.*;
import java.util.*;

public class Heapsort extends GFrame {
  TreeView tv;
  ArrayView av;
  BarView bv;

  public Heapsort(GAlgorithm alg) {
    super("Heapsort");
    setSize(600, 500);

    // Definition der Hauptkomponente
    GPanel gp = new GPanel();
```

```
    gp.setLayout(new GridLayout(2, 1));

    GPanel top = new GPanel(new GridLayout(1, 2));
    tv = new TreeView(((HeapsortAlgorithm) alg).A);
    top.add(tv);
    bv = new BarView(((HeapsortAlgorithm)alg).A);
    top.add(bv);
    gp.add(top);
    av = new ArrayView(((HeapsortAlgorithm) alg).A);
    gp.add(av);

    add(gp);
    setVisible(true);
    toBack();

    // Melde Sichten bei der Kontrolle an
    GEventControl control = alg.getControl();
    control.addIEListener(tv);
    control.addIEListener(av);
    control.addIEListener(bv);
  }
}
```

B Deskriptive Statistik der Evaluation

Dieser Anhang enthält die Darstellung und tabellarische Auswertung dreier Fragebögen, die im Rahmen der Evaluation des GANIFA-Systems im November 2000 an insgesamt 118 Versuchspersonen verteilt wurden (vgl. Kapitel 8). Die Befragung umfaßte

- einen Leistungstest (Abschnitt B.1),
- eine Bewertung der Lehrmittel (Abschnitt B.2) und
- Angaben zum eigenen Lernverhalten (Abschnitt B.3).

Alle Fragen sind in *geneigter Schrift* gesetzt, um sie vom Text der jeweils unmittelbar nachfolgenden Auswertung eindeutig unterscheiden zu können.

B.1 Leistungstest

Hinweise Die Fragen sind durchnumeriert und mit einem Präfix versehen: W steht für Wissensfrage und T für Transferfrage. Bei allen Multiple-Choice-Fragen ist jeweils genau eine Antwort richtig. Nach jeder Fragestellung ist die zu erreichende Punktzahl in Klammern angegeben. Die nach jeder Frage folgende Auswertungstabelle gibt die summativen Häufigkeiten der Antworten für jede Gruppe an. Hierbei ist der Buchstabe P eine Abkürzung für die erreichten Punkte, f_G bezeichnet die Häufigkeit, in der das Merkmal für die Gruppe $G \in \{1, 2, 3, 4\}$ in der Stichprobe auftrat, und $\%_G$ den entsprechenden Anteil in Prozent. Prozentwerte sind auf ganze Zahlen gerundet.

Fragen und Auswertung

W1. Seien r_1,r_2 und r reguläre Ausdrücke, Σ ein endliches Alphabet und ϵ das leere Wort. Reguläre Ausdrücke sind:

a) $\epsilon, a \in \Sigma, r_1 r_2, r_1|r_2, r_1 * r_2, (r), \emptyset$

b) $\epsilon, a \in \Sigma, r_1 r_2, r_1|r_2, r^*, (r), \emptyset$

c) $a \in \Sigma, r_1 r_2, r_1 * r_2, r_1|r_2, (r), \emptyset$

d) $a \in \Sigma, r_1 * r_2, r_1|r_2, (r), \emptyset$

e) $\epsilon, a \in \Sigma, r_1 * r_2, r_1|r_2, r, \emptyset$

Nichts sonst ist ein regulärer Ausdruck. *(1 Punkt)*

Gruppe	0 P		1 P		Σ
	f_G	$\%_G$	f_G	$\%_G$	
1 – AdlA	6	21	23	79	29
2 – GaniFA	7	23	23	77	30
3 – Text	2	7	28	93	30
4 – Vorlesung	9	31	20	69	29
Σ	24	20	94	80	118

W2. Welcher reguläre Ausdruck entspricht Bezeichnern, wie sie in der Lerneinheit definiert wurden? *(1 Punkt)*

a) $bu(bu|zi)$ *c)* $bu(bu|zi)^*$ *e)* $zi(bu|zi^*)$

b) $bu(bu|zi^*)$ *d)* $zi(bu|zi)$ *f)* $zi(bu|zi)^*$

Gruppe	0 P		1 P		Σ
	f_G	$\%_G$	f_G	$\%_G$	
1 – AdlA	4	14	25	86	29
2 – GaniFA	8	27	22	73	30
3 – Text	4	13	26	87	30
4 – Vorlesung	15	52	14	48	29
Σ	31	26	87	74	118

W3. Was ist am endlichen Automaten „endlich“? *(1 Punkt)*

Gruppe	0 P		1 P		Σ
	f_G	$\%_G$	f_G	$\%_G$	
1 – AdlA	6	21	23	79	29
2 – GaniFA	6	20	24	80	30
3 – Text	8	27	22	73	30
4 – Vorlesung	11	38	18	62	29
Σ	31	26	87	74	118

W4. Welche der folgenden Worte gehören in die durch den regulären Ausdruck ab^*a *beschriebene Sprache?* *(1 Punkt)*

a) $\epsilon, a, aba, abaa, abaaa$ *d)* $aa, aba, abba, abbba$

b) $aba, abab, abb$ *e)* ϵ, aa, ab, aba

c) $aba, abab, abb, abba$ *f)* $\epsilon, a, ab, aba, abba$

Gruppe	0 P		1 P		Σ
	f_G	$\%_G$	f_G	$\%_G$	
1 – AdlA	1	3	28	97	29
2 – GaniFA	5	17	25	83	30
3 – Text	5	17	25	83	30
4 – Vorlesung	4	14	25	86	29
Σ	15	13	103	87	118

W5. Welche der folgenden Worte gehören in die durch den regulären Ausdruck $(ab)^*$ beschriebene Sprache? (1 Punkt)

a) $\epsilon, a, aba, abaa, abaaa$

b) $ab, abab, abababa$

c) $aba, abab, abb, abba$

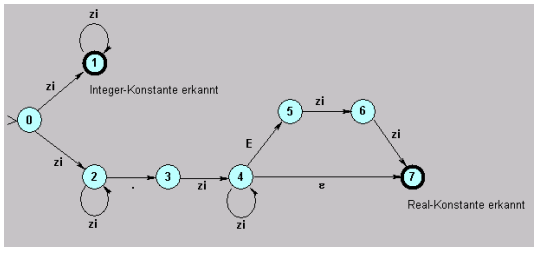
d) $\epsilon, ab, abab, ababab$

e) $\epsilon, b, bab, babab$

f) $\epsilon, a, ab, abab$

Gruppe	0 P		1 P		Σ
	f_G	$\%_G$	f_G	$\%_G$	
1 – AdlA	4	14	25	86	29
2 – GaniFA	4	14	26	86	30
3 – Text	4	14	26	86	30
4 – Vorlesung	6	21	23	79	29
Σ	18	15	100	85	118

W6. Gib jeweils den Weg, d. h. die Folge von Zuständen an, die der Automat beim Lesen der gegebenen Eingabe durchläuft.



W6.1) 3.14 *($\frac{1}{2}$ Punkt)*

W6.2) 24 *($\frac{1}{2}$ Punkt)*

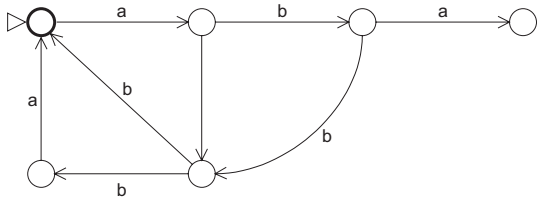
W6.3) $0.75E43$ *($\frac{1}{2}$ Punkt)*

W6.4) $0.75.E432$ *($\frac{1}{2}$ Punkt)*

Gruppe	W6.1				W6.2				Σ
	0 P		1/2 P		0 P		1/2 P		
	f_G	$\%_G$	f_G	$\%_G$	f_G	$\%_G$	f_G	$\%_G$	
1 – AdlA	10	34	19	66	4	14	25	86	29
2 – GaniFA	7	23	23	77	3	10	27	90	30
3 – Text	3	10	27	90	2	7	28	93	30
4 – Vorlesung	8	28	21	72	2	7	27	93	29
Σ	28	24	90	76	11	9	107	91	118

Gruppe	W6.3				W6.4				Σ
	0 P		1/2 P		0 P		1/2 P		
	f_G	$\%_G$	f_G	$\%_G$	f_G	$\%_G$	f_G	$\%_G$	
1 – AdlA	5	17	24	83	12	41	17	59	29
2 – GaniFA	3	10	27	90	13	43	17	57	30
3 – Text	1	3	29	97	8	27	22	73	30
4 – Vorlesung	4	14	25	86	9	31	20	69	29
Σ	13	11	105	89	42	36	76	64	118

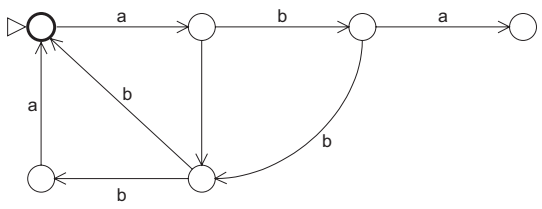
T1. Welches der angegebenen Worte wird von dem im folgenden Übergangsdiagramm dargestellten endlichen Automaten akzeptiert? (1 Punkt)



a) aa
b) abaabaa
c) abb
d) abaabbb
e) baab
f) abbbaa

Gruppe	0 P		1 P		Σ
	f_G	$\%_G$	f_G	$\%_G$	
1 – AdlA	7	24	22	76	29
2 – GaniFA	7	23	23	77	30
3 – Text	6	20	24	80	30
4 – Vorlesung	10	34	19	66	29
Σ	30	25	88	75	118

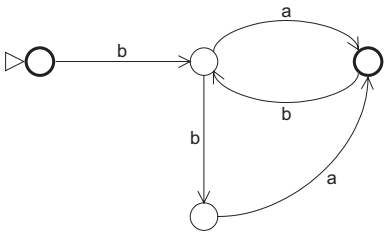
T2. Welcher reguläre Ausdruck beschreibt die Sprache, die der im folgenden Übergangsdiagramm dargestellte endliche Automat akzeptiert? (1 Punkt)



a) $(a(b|bbb)a)^*$
b) $(a(\epsilon|bb)(b|ba))^*$
c) $(aba)^*(abbba)^*(ab|b)$
d) $((abb)|(ab)|(aba))^*$
e) $(a(bb|\epsilon|ba))^*$
f) $a(abb|bbb|ba)^*$

Gruppe	0 P		1 P		Σ
	f_G	$\%_G$	f_G	$\%_G$	
1 – AdlA	15	52	14	48	29
2 – GaniFA	17	57	13	43	30
3 – Text	9	30	21	70	30
4 – Vorlesung	15	52	14	48	29
Σ	56	47	62	53	118

T3. Welches der angegebenen Worte wird von dem im folgenden Übergangsdiagramm dargestellten endlichen Automaten akzeptiert? (1 Punkt)



a) babbabb

b) bbababababba

c) babbab

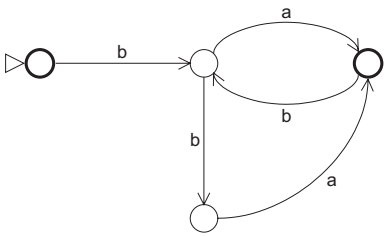
d) abbabab

e) bbabaaba

f) bababbaabba

Gruppe	0 P		1 P		Σ
	f_G	$\%_G$	f_G	$\%_G$	
1 – AdlA	1	3	28	97	29
2 – GaniFA	2	7	28	93	30
3 – Text	2	7	28	93	30
4 – Vorlesung	2	7	27	93	29
Σ	7	6	111	94	118

T4. Welcher reguläre Ausdruck beschreibt die Sprache, die der im folgenden Übergangsdiagramm dargestellte endliche Automat akzeptiert? (1 Punkt)



a) $b(aba|aa)(baba|baa)^*$

b) $(ba|bba)(ba|a)^*$

c) $\epsilon|(b(a|ba)((a|ba)(\epsilon|b))^*)$

d) $b(a|ba)((ba|a)b)^*$

e) $b(ab)^*(bab)^*$

f) $(b(a|ba)(b(ba|a))^*)|\epsilon$

Gruppe	0 P		1 P		Σ
	f_G	$\%_G$	f_G	$\%_G$	
1 – AdlA	11	38	18	62	29
2 – GaniFA	13	43	17	57	30
3 – Text	11	37	19	63	30
4 – Vorlesung	12	41	17	59	29
Σ	47	40	71	60	118

T5. Welche der nachstehenden Wortfolgen gehören in die durch den regulären Ausdruck $(a(b|\epsilon)c^*b)|(bc^*)$ beschriebene Sprache? (1 Punkt)

a) $b, ac, acb, abb, acccb$

b) $b, ac, bcc, abb, acccb$

c) $b, bc, bcc, ab, acccb$

d) $b, ac, bcc, abb, acccb$

e) $a, ab, bc, abccccb$

f) $abcccb, bccc, bcccb, abccc$

Gruppe	0 P		1 P		Σ
	f_G	$\%_G$	f_G	$\%_G$	
1 – AdlA	14	48	15	52	29
2 – GaniFA	13	43	17	57	30
3 – Text	12	40	18	60	30
4 – Vorlesung	21	72	8	28	29
Σ	60	51	58	49	118

T6. Welche der nachstehenden Wortfolgen gehören in die durch den regulären Ausdruck $(a|(b|c^*))|(b(c^*|a))$ beschriebene Sprache? (1 Punkt)

a) $a, b, c, bccca$

b) $acccc, cccc, bccc$

c) $\epsilon, bc, bcc, ab, acccb$

d) $b, bc, bcc, bcca, acccb$

e) a, b, c, bc, ba, ab

f) $\epsilon, a, b, c, bc, ba$

Gruppe	0 P		1 P		Σ
	f_G	$\%_G$	f_G	$\%_G$	
1 – AdlA	11	38	18	62	29
2 – GaniFA	8	27	22	73	30
3 – Text	13	43	17	57	30
4 – Vorlesung	15	52	14	48	29
Σ	47	40	71	60	118

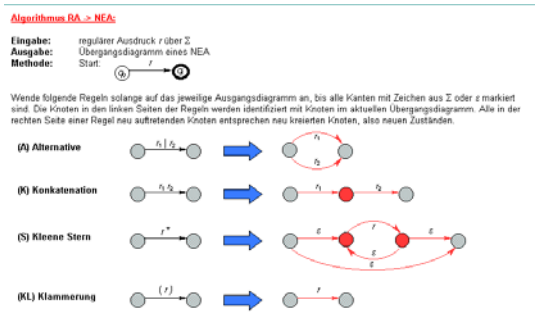
T7. Sei Z0 der Startzustand und Z2 ein Endzustand. *(2 Punkte)*

	a	b	ϵ
$Z0$	$\{Z1, Z2\}$	$Z0$	$Z2$
$Z1$	–	$Z2$	–
$Z2$	–	–	–

Gib einen regulären Ausdruck an, der die von dem durch die links stehende Übergangstabelle gegebenen Automaten akzeptierte Sprache beschreibt.

Gruppe	0 P		1 P		2 P		Σ
	f_G	$\%_G$	f_G	$\%_G$	f_G	$\%_G$	
1 – AdlA	15	52	4	14	10	34	29
2 – GaniFA	13	43	9	30	8	27	30
3 – Text	17	56	8	27	5	17	30
4 – Vorlesung	17	59	3	10	9	31	29
Σ	62	53	24	20	32	27	118

T8. Die Lerneinheit hat Dir folgendes Verfahren zur Konstruktion des Übergangsdiagramms eines NEA's aus einem regulären Ausdruck vorgestellt. *(2 Punkte)*



Konstruiere nun aus dem folgenden regulären Ausdruck das korrespondierende Übergangsdiagramm: $ma^*(oa|ao)a^*m$.

Gruppe	0 P		1 P		2 P		Σ
	f_G	$\%_G$	f_G	$\%_G$	f_G	$\%_G$	
1 – AdlA	9	31	-	-	20	69	29
2 – GaniFA	8	27	6	20	16	53	30
3 – Text	5	17	4	13	21	70	30
4 – Vorlesung	11	38	4	14	14	48	29
Σ	33	28	14	12	71	60	118

T9. Sei r ein regulärer Ausdruck. r^* beschreibt die Sprache $L(r^*) = \{\omega^n | \omega \in L(r), n \geq 0\}$. Nun definieren wir folgende Notation: der reguläre Ausdruck r^+ beschreibt die Sprache $L(r^+) = \{\omega^n | \omega \in L(r), n \geq 1\}$. Zum Beispiel: $L(a^+) = \{a, aa, aaa, \ldots\}$. Gib eine Konstruktionsregel für ein Übergangsdiagramm eines NEA an, der die oben definierte Sprache $L(r^+)$ akzeptiert. *(1 Punkt)*

Gruppe	0 P		1 P		Σ
	f_G	$\%_G$	f_G	$\%_G$	
1 – AdlA	17	59	12	41	29
2 – GaniFA	13	43	17	57	30
3 – Text	18	60	12	40	30
4 – Vorlesung	19	66	10	34	29
Σ	67	57	51	43	118

T10. Gib einen regulären Ausdruck für die Sprache an, die genau die Worte enthält, die in der durch den regulären Ausdruck $a(a|b)^*$ und der durch den regulären Ausdruck $(b|a^*)b^*$ beschriebenen Sprache enthalten sind, d. h. der reguläre Ausdruck soll die Schnittmenge der beiden Sprachen beschreiben. (2 Punkte)

Gruppe	0 P		1 P		2 P		Σ
	f_G	$\%_G$	f_G	$\%_G$	f_G	$\%_G$	
1 – AdlA	23	79	2	7	4	14	29
2 – GaniFA	19	64	4	13	7	23	30
3 – Text	26	87	1	3	3	10	30
4 – Vorlesung	23	79	4	14	2	7	29
Σ	91	77	11	9	16	14	118

B.2 Analyse des Bewertungsfragebogens

Hinweise Die Fragen sind durchnumeriert und mit einem Präfix B versehen. Für jede Gruppe ist ein eigener Unterabschnitt vorgesehen, da sich die Fragen je nach Lehrmethode unterscheiden. Die nach jeder Frage folgende Auswertungstabelle gibt die summativen Häufigkeiten der angekreuzten Werte von Rating-Skalen an. Hierbei steht die Abkürzung „k.A." für eine fehlende Angabe und $\overline{x}$ für den Mittelwert der entsprechenden Rating-Skala. Weiterhin bezeichnen die Symbole $\top$ und $\bot$ das obere bzw. untere Ende der Skala. Hinter den Aussagen zu den offenen Fragen steht in Klammern die Anzahl der Personen, die diese Aussage vermerkt haben. Den Versuchspersonen wurde vor der Beantwortung der Fragen die nachfolgende Anleitung gegeben:

Instruktion: Für die Beantwortung der folgenden Fragen stehen Dir sogenannte Rating-Skalen zur Verfügung, d. h. Du kannst Deine Antwort zwischen „0" für die Antwortvorgabe auf der linken Seite und „10" für die entgegengesetzte Antwortvorgabe auf der rechten in Einer-Schritten abstufen. Achte darauf, daß Du nicht versehentlich die Seiten vertauschst und „falsch herum" ankreuzt! Bei einigen Fragen gibt es anstatt der Rating-Skalen ein leeres Kästchen. Trage hier Deine Antwort mit Deinen eigenen Worten kurz, aber verständlich in Stichpunkten ein.

Beispiel: Nimmst Du ungern oder gern an Untersuchungen zu neuen Lehrmitteln teil?

sehr ungern	*0 – 1 – 2 – 3 – 4 – 5 – 6 – 7 – X – 9 – 10*	*sehr gern*
Wenn Du relativ gern an solchen Versuchen teilnimmst, kreuzt Du z. B. die 8 an.		

Gruppe 1 – ADLA

29 Probanden

B1. War das Thema eher leicht oder eher schwer zu verstehen?

$\perp$	0	1	2	3	4	5	6	7	8	9	10	$\top$	k.A.	$\overline{x}$
sehr schwer	-	-	1	1	1	6	4	5	6	3	1	sehr leicht	1	6,5

B2. Hat die Lernsoftware das Verständnis des Themas eher erleichtert oder eher nicht erleichtert?

$\perp$	0	1	2	3	4	5	6	7	8	9	10	$\top$	k.A.	$\overline{x}$
überhaupt nicht erleichtert	-	1	2	2	6	4	4	7	2	1	-	sehr stark erleichtert	-	5,3

B3. Welche Eigenschaften des Systems halfen, den Lernstoff besser zu verstehen?

- die Beispiele (9)
- die Animationssequenzen (8)
- selbstbestimmbares Lerntempo (3)
- zusätzliche Definitionen (3)
- die Graphiken (5)
- schrittweise Erklärung (2)
- gute didaktische Aufbereitung (3)

B4. Welche Eigenschaften des Systems erschwerten das Lernen und Verstehen?

- Animationen/Videos liefen zu schnell ab (8)
- Animationen sind nicht rückwärts abspielbar (1)
- Definitionen sind zu formal (3)
- keine Möglichkeit, Rückfragen zu stellen (2)
- Einzelschrittmodus nicht vorhanden (4)
- die Erklärungen des Automatenmodells (2)
- zu wenig Beispiele (1)
- das Lernen am Bildschirm (2)

B5. Wie bewertest Du im Vergleich mit dem Lehrbuch „Hardware Design“ von Keller und Paul diese Software? Im Vergleich zu diesem Lehrbuch ist das Lernen mit dieser Software ...

$\perp$	0	1	2	3	4	5	6	7	8	9	10	$\top$	k.A.	$\overline{x}$
sehr viel schwieriger	-	-	-	4	2	2	6	1	6	3	-	sehr viel einfacher	5	6,2
sehr viel unangenehmer	-	-	2	3	1	2	3	4	3	4	2	sehr viel angenehmer	5	6,4
sehr viel uneffektiver	-	-	-	3	3	8	2	4	2	2	-	sehr viel effektiver	5	5,6
sehr viel uninteressanter	-	1	-	-	2	4	3	1	9	2	2	sehr viel interessanter	5	6,8
sehr viel unmotivierender	-	1	-	1	2	4	4	3	6	2	1	sehr viel motivierender	5	6,4
sehr viel anstrengender	-	1	3	2	-	5	4	2	4	1	2	sehr viel weniger anstrengend	5	5,7
sehr viel zeitaufwendiger	-	-	1	2	-	5	7	3	2	1	3	sehr viel zeitsparender	5	6,3

5 Teilnehmer kannten das Lehrbuch „Hardware Design“ von Keller und Paul nicht.

B6. Würdest Du ein solches Programm zu Hause als Ergänzung zu Deinen Lehrveranstaltungen und Prüfungen verwenden?

$\perp$	0	1	2	3	4	5	6	7	8	9	10	$\top$	k.A.	$\overline{x}$
auf keinen Fall	2	-	2	1	2	1	1	4	7	3	6	auf jeden Fall	-	6,8

B7. Warum würdest Du dieses Programm zu Hause benutzen bzw. warum würdest Du es nicht tun?

- *Ich würde es zur Ergänzung benutzen, weil*
  - die Animationen zum Verständnis beitragen (4)
  - es übersichtlich und motivierend ist (4)
  - es als Ergänzung zur Vorlesung hilfreich ist (4)
  - die Beispiele gut sind (2)
  - schnelles Hin- und Herspringen möglich ist (1)
  - es zur Wiederholung geeignet ist (1)
  - die deutsche Sprache das Verständnis erleichtert (1)
  - viele Quellen (Vorlesung, Buch etc.) sich ergänzen (1)
- *Ich würde es nicht zur Ergänzung benutzen, weil*
  - das Lernen am Bildschirm anstrengend ist (2)

- zu wenig Interaktion möglich ist (1)
- Bücher besser zum Lernen sind (2)
- die Informationen schlecht aufbereitet sind (3)

B8. Hat Dir die Steuerung der Animationen Schwierigkeiten oder keine Schwierigkeiten bereitet?

$\perp$	0	1	2	3	4	5	6	7	8	9	10	$\top$	k.A.	$\overline{x}$
große Schwierigkeiten	1	-	-	1	2	-	1	2	3	11	8	gar keine Schwierigkeiten	-	8,1

B9. Wenn ja, womit hattest Du Probleme?

- Die Videos/Animationen liefen zu schnell ab (5)
- Die Möglichkeit, die Animationen schrittweise ablaufen zu lassen, mußte erst entdeckt werden (2)
- Man konnte keinen Schritt zurück machen (3)
- Man konnte die Animationen nicht stoppen (3)

B10. War der Aufruf der Beispielanimationen schwierig oder einfach zu finden?

$\perp$	0	1	2	3	4	5	6	7	8	9	10	$\top$	k.A.	$\overline{x}$
sehr schwierig	-	-	-	1	-	1	1	5	10	4	7	überhaupt nicht schwierig	-	8,1

B11. Durch Anklicken gewisser „Hotwords“ wurden deren Definitionen aufgerufen. Hattest Du damit Schwierigkeiten?

$\perp$	0	1	2	3	4	5	6	7	8	9	10	$\top$	k.A.	$\overline{x}$
große Schwierigkeiten	-	-	1	-	-	-	-	-	4	7	17	gar keine Schwierigkeiten	-	9,2

B12. Hat Dir das Design der Seiten und Beispiele gefallen?

$\perp$	0	1	2	3	4	5	6	7	8	9	10	$\top$	k.A.	$\overline{x}$
überhaupt nicht gefallen	1	-	1	-	-	4	5	11	2	5	-	sehr gefallen	-	6,6

Gruppe 2 – GANIFA

30 Probanden

B1. War das Thema eher leicht oder eher schwer zu verstehen?

$\perp$	0	1	2	3	4	5	6	7	8	9	10	$\top$	k.A.	$\overline{x}$
sehr schwer	-	-	-	2	1	5	6	5	9	2	-	sehr leicht	-	6,6

B2. Hat die Lernsoftware das Verständnis des Themas eher erleichtert oder eher nicht erleichtert?

$\perp$	0	1	2	3	4	5	6	7	8	9	10	$\top$	k.A.	$\overline{x}$
überhaupt nicht erleichtert	-	2	3	6	2	2	4	5	3	1	2	sehr stark erleichtert	-	5,2

B3. Welche Eigenschaften des Systems halfen, den Lernstoff besser zu verstehen?

- die Beispiele (2)
- die Animationen (8)
- das Herumspielen mit Spezifikationen (2)
- zusätzliche Definitionen (7)
- die Graphiken (4)
- gute didaktische Aufbereitung (4)

B4. Welche Eigenschaften des Systems erschwerten das Lernen und Verstehen?

- Animationen schlecht dokumentiert (1)
- Applets lenken vom Stoff ab (2)
- Definitionen sind zu formal (3)
- keine Lehrperson und Feedback (1)
- Einzelschrittmodus nicht vorhanden (1)
- die graphische Aufmachung (5)
- Eingabe der regulären Ausdrücke nicht intuitiv (2)
- die Gliederung des Textes (1)
- das Lernen am Bildschirm (2)

B5. Wie bewertest Du im Vergleich mit dem Lehrbuch „Hardware Design“ von Keller und Paul diese Software? Im Vergleich zu diesem Lehrbuch ist das Lernen mit dieser Software ...

$\perp$	0	1	2	3	4	5	6	7	8	9	10	$\top$	k.A.	$\overline{x}$
sehr viel schwieriger	1	-	3	2	2	4	5	3	4	1	1	sehr viel einfacher	4	5,1
sehr viel unangenehmer	-	-	2	1	2	2	7	6	4	-	2	sehr viel angenehmer	4	5,3
sehr viel uneffektiver	1	-	3	4	3	2	4	4	3	1	1	sehr viel effektiver	4	4,7
sehr viel uninteressanter	1	-	-	2	2	5	3	4	4	3	2	sehr viel interessanter	4	5,7
sehr viel unmotivierender	1	-	1	2	-	5	5	5	2	3	2	sehr viel motivierender	4	5,3
sehr viel anstrengender	1	-	1	1	3	9	2	3	2	2	2	sehr viel weniger anstrengend	4	4,7
sehr viel zeitaufwendiger	1	-	1	2	-	6	6	3	4	1	2	sehr viel zeitsparender	4	5,2

4 Teilnehmer kannten das Lehrbuch „Hardware Design“ von Keller und Paul nicht.

B6. Würdest Du ein solches Programm zu Hause als Ergänzung zu Deinen Lehrveranstaltungen und Prüfungen verwenden?

$\perp$	0	1	2	3	4	5	6	7	8	9	10	$\top$	k.A.	$\overline{x}$
auf keinen Fall	3	-	-	3	2	1	4	1	5	4	7	auf jeden Fall	-	6,6

B7. Warum würdest Du dieses Programm zu Hause benutzen bzw. warum würdest Du es nicht tun?

- *Ich würde es zur Ergänzung benutzen, weil*
  - die Animationen zum Verständnis beitragen (3)
  - es übersichtlich und motivierend ist (3)
  - es als Ergänzung zur Vorlesung hilfreich ist (5)
  - die Beispiele gut sind (1)
  - schnelles Hin- und Herspringen möglich ist (1)
  - es zur Wiederholung geeignet ist (1)
  - die deutsche Sprache das Verständnis erleichtert (1)
  - viele Quellen (Vorlesung, Buch etc.) sich ergänzen (1)
- *Ich würde es nicht zur Ergänzung benutzen, weil*
  - das Lernen am Bildschirm anstrengend ist (1)
  - zuwenig Interaktivität vorhanden ist (1)
  - die Aufmachung zuviel ablenkt (1)
  - Bücher besser zum Lernen sind (4)
  - die Graphiken und Informationen schlecht aufbereitet sind (1)

B8. Hat Dir die Steuerung der Animationen Schwierigkeiten oder keine Schwierigkeiten bereitet?

$\perp$	0	1	2	3	4	5	6	7	8	9	10	$\top$	k.A.	$\overline{x}$
große Schwierigkeiten	2	-	4	1	1	1	-	2	4	6	7	gar keine Schwierigkeiten	2	6,8

B9. Wenn ja, womit hattest Du Probleme?

- Die Animationen liefen zu schnell ab (2)
- Mit der Spezifikation der regulären Ausdrücke (4)
- Zuviele Pfeile im Automaten (1)
- Symbolik der Farben nicht verstanden (4)
- Man konnte keinen Schritt zurück machen (1)
- Die Fußnoten waren zu kompliziert (1)

B10. War der Aufruf der Beispielanimationen schwierig oder einfach zu finden?

$\perp$	0	1	2	3	4	5	6	7	8	9	10	$\top$	k.A.	$\overline{x}$
sehr schwierig	-	-	-	2	-	3	1	2	4	9	9	überhaupt nicht schwierig	-	8,1

B11. Durch Anklicken gewisser „Hotlinks“ wurden deren Definitionen aufgerufen. Hattest Du damit Schwierigkeiten?

$\perp$	0	1	2	3	4	5	6	7	8	9	10	$\top$	k.A.	$\overline{x}$
große Schwierigkeiten	-	-	-	-	-	-	1	-	3	4	21	gar keine Schwierigkeiten	1	9,5

B12. Hat Dir das Design der Seiten und Beispiele gefallen?

$\perp$	0	1	2	3	4	5	6	7	8	9	10	$\top$	k.A.	$\overline{x}$
überhaupt nicht gefallen	-	-	1	3	2	1	1	3	11	3	5	sehr gefallen	-	7,2

Gruppe 3 – Text

30 Probanden

B1. War das Thema eher leicht oder eher schwer zu verstehen?

$\perp$	0	1	2	3	4	5	6	7	8	9	10	$\top$	k.A.	$\overline{x}$
sehr schwer	-	-	-	-	-	6	4	4	7	4	5	sehr leicht	-	7,5

B2. Hat der Text das Verständnis des Themas eher erleichtert oder eher nicht erleichtert?

$\perp$	0	1	2	3	4	5	6	7	8	9	10	$\top$	k.A.	$\overline{x}$
überhaupt nicht erleichtert	-	-	-	2	-	4	4	5	9	6	-	sehr stark erleichtert	-	7,0

B3. Welche Eigenschaften des Textes halfen, den Lernstoff besser zu verstehen?

- die Beispiele (19)
- Aufbau des Lehrstoffes (1)
- die Erklärungen (1)
- die formale Beschreibung (2)
- zusätzliche Definitionen (1)
- die Graphiken (11)
- gute didaktische Aufbereitung (3)

B4. Welche Eigenschaften des Textes erschwerten das Lernen und Verstehen?

- Abweichung von der schon erlernten Terminologie (2)
- unnötige Beispiele (1)
- Definitionen sind zu formal (10)
- Querverweise zu den Definitionen (4)
- zu lange Sätze (1)
- die Gliederung des Textes (1)
- zu knappe Erklärungen (1)
- zu lange Erklärungen (2)

B5. Wie bewertest Du im Vergleich mit dem Lehrbuch „Hardware Design“ von Keller und Paul diesen Text? Im Vergleich zu diesem Lehrbuch ist das Lernen mit diesem Text ...

$\perp$	0	1	2	3	4	5	6	7	8	9	10	$\top$	k.A.	$\overline{x}$
sehr viel schwieriger	1	-	-	3	-	3	3	7	6	1	5	sehr viel einfacher	1	6,8
sehr viel unangenehmer	-	-	-	3	3	4	3	5	5	2	4	sehr viel angenehmer	1	6,6
sehr viel uneffektiver	-	-	1	-	2	11	1	4	5	3	2	sehr viel effektiver	1	6,4
sehr viel uninteressanter	1	1	2	-	3	8	2	3	4	2	3	sehr viel interessanter	1	5,9
sehr viel unmotivierender	1	-	1	-	3	11	2	3	3	2	3	sehr viel motivierender	1	6,0
sehr viel anstrengender	-	1	-	1	-	5	3	5	5	3	6	sehr viel weniger anstrengend	1	7,2
sehr viel zeitaufwendiger	-	-	2	1	3	3	3	3	5	4	5	sehr viel zeitsparender	1	6,9

1 Teilnehmer kannte das Lehrbuch „Hardware Design“ von Keller und Paul nicht.

B11. Durch Verweise bei gewissen „Hotwords“ wurde auf deren Definitionen im Anhang aufmerksam gemacht. Hattest Du damit Schwierigkeiten?

$\perp$	0	1	2	3	4	5	6	7	8	9	10	$\top$	k.A.	$\overline{x}$
große Schwierigkeiten	-	2	-	4	2	4	1	1	2	3	11	gar keine Schwierigkeiten	-	6,9

B12. Hat Dir das Design der Seiten und Beispiele gefallen?

$\perp$	0	1	2	3	4	5	6	7	8	9	10	$\top$	k.A.	$\overline{x}$
überhaupt nicht gefallen	1	1	-	2	1	5	2	6	8	4	-	sehr gefallen	-	6,3

Gruppe 4 – Vorlesung

29 Probanden

B1. War das Thema eher leicht oder eher schwer zu verstehen?

$\perp$	0	1	2	3	4	5	6	7	8	9	10	$\top$	k.A.	$\overline{x}$
sehr schwer	-	-	1	1	3	4	4	4	6	4	1	sehr leicht	1	6,5

B2. Hat die Vorlesung das Verständnis des Themas eher erleichtert oder eher nicht erleichtert?

$\perp$	0	1	2	3	4	5	6	7	8	9	10	$\top$	k.A.	$\overline{x}$
überhaupt nicht erleichtert	1	1	2	5	2	6	4	5	3	-	-	sehr stark erleichtert	-	4,9

B3. Welche Eigenschaften der Vorlesung halfen, den Lernstoff besser zu verstehen?

- die Beispiele (10)
- die verbalen Erklärungen (2)
- der Tafelanschrieb (1)
- die Tafelbilder (5)
- das gleichzeitige Konsumieren von Erklärungen und Tafelbildern (2)
- klare Sprache und Ausdrucksform (2)
- der anschließende Leistungstest (1)

B4. Welche Eigenschaften der Vorlesung erschwerten das Lernen und Verstehen?

- zu kleine Schrift (3)
- kein Skript (1)
- Definitionen sind zu formal (3)
- zu starres Anschreiben (2)
- gleichzeitiges Mitschreiben und Verstehen (1)
- zu schnelles Vorgehen (15)
- das Verbot, Zwischenfragen zu stellen (2)
- zu viele Informationen auf einmal (3)

B5. Wie bewertest Du im Vergleich mit dem Lehrbuch „Hardware Design“ von Keller und Paul diese Vorlesung? Im Vergleich zu diesem Lehrbuch ist das Lernen mit dieser Vorlesung ...

$\perp$	0	1	2	3	4	5	6	7	8	9	10	$\top$	k.A.	$\overline{x}$
sehr viel schwieriger	2	-	-	2	1	9	7	6	-	-	1	sehr viel einfacher	1	5,3
sehr viel unangenehmer	1	-	2	2	1	7	3	6	4	1	1	sehr viel angenehmer	1	5,6
sehr viel uneffektiver	-	1	-	-	4	8	10	2	3	-	-	sehr viel effektiver	1	5,5
sehr viel uninteressanter	-	-	1	4	3	8	2	5	2	-	2	sehr viel interessanter	2	5,5
sehr viel unmotivierender	-	-	2	1	3	5	5	8	3	-	1	sehr viel motivierender	1	5,9
sehr viel anstrengender	1	1	2	3	1	6	4	5	5	-	-	sehr viel weniger anstrengend	1	5,3
sehr viel zeitaufwendiger	-	1	-	1	2	7	4	6	1	4	1	sehr viel zeitsparender	2	6,2

1 Teilnehmer kannte das Lehrbuch „Hardware Design“ von Keller und Paul nicht.

B12. Hat Dir das Design der Tafelbilder und Beispiele gefallen?

⊥	0	1	2	3	4	5	6	7	8	9	10	⊤	k.A.	$\overline{x}$
überhaupt nicht gefallen	-	1	1	3	3	4	3	5	4	3	1	sehr gefallen	1	5,9

B.3 Auswertung des Fragebogens zum Lernverhalten

Hinweise Die Fragen sind durchnumeriert und mit einem Präfix *S* versehen. Die nach jeder Frage folgende Auswertungstabelle gibt die summativen Häufigkeiten der Antworten für jede Gruppe an. Hierbei symbolisieren die Abkürzungen „k.A.“ eine fehlende Angabe und „A“ eine Antwort. Hinter den Aussagen zu den offenen Fragen stehen in Klammern die Anzahl der Personen, die diese Aussage vermerkt haben, sowie die Gruppen, zu denen diese Personen gehörten. Den Versuchspersonen wurde vor der Beantwortung der Fragen die nachfolgende Anleitung gegeben:

Instruktion: Zum Abschluß möchten wir Dich bitten, uns noch ein paar Angaben zu Deiner Person zu machen. Sie sollen uns helfen, die Stichprobe als Ganzes genauer zu charakterisieren, damit wir unsere Ergebnisse besser mit anderen Forschungsprojekten vergleichen können. Vor allem können wir damit die statistische Auswertung verfeinern, d. h. die Qualität der Ergebnisse wesentlich verbessern. Wir möchten Dich ganz besonders bitten, ehrlich und präzise zu antworten.

Fragen und Auswertung

S1. Wie alt bist Du?

Gruppe	20-25	26-30	≥ 31	k.A.	Σ
1 – AdlA	20	6	-	3	29
2 – GaniFA	26	3	-	1	30
3 – Text	27	2	1	-	30
4 – Vorlesung	28	1	-	-	29
Σ	101	12	1	4	118

S2. Geschlecht?

Gruppe	männlich	weiblich	k.A.	Σ
1 – AdlA	26	1	2	29
2 – GaniFA	23	7	-	30
3 – Text	25	5	-	30
4 – Vorlesung	25	4	-	29
Σ	99	17	2	118

S3. Im wievielten Semester studierst Du Informatik?

Gruppe	1-3	4-6	7-9	≥ 10	k.A.	Σ
1 – AdlA	19	4	2	2	2	29
2 – GaniFA	20	8	2	-	-	30
3 – Text	22	4	2	1	1	30
4 – Vorlesung	19	7	3	-	-	29
Σ	80	23	9	3	3	118

S4. Hast Du Dich schon einmal mit den Themen beschäftigt, die in diesem Test vorkamen?

A1. Nein, ich kenne die Themen überhaupt nicht

A2. Ja, im Studium

A3. Ja, im Informatikunterricht in der Schule

A4. Ja, bei folgender Gelegenheit (bitte angeben [S5])

Gruppe	A1	A2	A3	A4	k.A.	Σ
1 – AdlA	1	23	2	1	2	29
2 – GaniFA	4	19	7	-	-	30
3 – Text	2	25	3	-	-	30
4 – Vorlesung	5	21	2	1	-	29
Σ	12	88	14	2	2	118

Gelegenheiten bei der Antwort A4:

- in einer Vorlesung (1, Gruppe 4)
- Programmierung mit Perl (1, Gruppe 1)

S6. Hast Du augenblicklich zu Hause einen eigenen Computer?

Gruppe	Ja	Nein	k.A.	Σ
1 – AdlA	27	-	2	29
2 – GaniFA	29	-	1	30
3 – Text	29	1	-	30
4 – Vorlesung	28	1	-	29
Σ	113	2	3	118

S7. In welchem Alter hattest Du Deinen ersten Computer bzw. wie alt warst Du, als der erste Computer in Deinem Haushalt angeschafft wurde (z. B. durch Geschwister oder Eltern), den Du regelmäßig benutzen durftest?

Gruppe	5	6	7	8	9	k.A.	Σ
1 – AdlA	-	-	-	1	2	26	29
2 – GaniFA	1	-	4	4	-	21	30
3 – Text	-	1	1	4	1	23	30
4 – Vorlesung	1	3	-	3	-	22	29
Σ	2	4	5	12	3	92	118

S8. Falls Dir privat ein Computer zur Verfügung steht, wofür benutzt Du Ihn? Bitte versuche möglichst genau die Stunden pro Woche abzuschätzen, die Du durchschnittlich mit den folgenden Dingen verbringst.

A1. Programmieren für das Studium

Gruppe	≤5	6-10	11-15	≥16	k.A.	Σ
1 – AdlA	26	-	-	1	2	29
2 – GaniFA	25	4	-	-	1	30
3 – Text	26	4	-	-	-	30
4 – Vorlesung	24	4	-	-	1	29
Σ	101	12	-	1	4	118

A2. Programmieren für einen Job

Gruppe	≤5	6-10	11-15	≥16	k.A.	Σ
1 – AdlA	22	1	1	3	2	29
2 – GaniFA	23	3	1	2	1	30
3 – Text	24	3	2	1	-	30
4 – Vorlesung	22	3	-	3	1	29
Σ	91	10	4	9	4	118

A3. Programmieren für private Dinge/Hobby

Gruppe	≤5	6-10	11-15	≥16	k.A.	Σ
1 – AdlA	24	3	-	-	2	29
2 – GaniFA	25	4	-	-	1	30
3 – Text	25	2	3	-	-	30
4 – Vorlesung	26	-	2	-	1	29
Σ	100	9	5	-	4	118

A4. Informationssuche (z. B. WWW, Datenbanken)

Gruppe	≤5	6-10	11-15	≥16	k.A.	Σ
1 – AdlA	17	8	1	1	2	29
2 – GaniFA	21	6	1	1	1	30
3 – Text	24	6	-	-	-	30
4 – Vorlesung	19	6	2	1	1	29
Σ	81	26	4	3	4	118

A5. Kommunikation (z. B. Email)

Gruppe	≤5	6-10	11-15	≥16	k.A.	Σ
1 – AdlA	24	2	-	-	3	29
2 – GaniFA	25	2	-	2	1	30
3 – Text	27	3	-	-	-	30
4 – Vorlesung	26	1	1	-	1	29
Σ	102	8	1	2	5	118

A6. Textverarbeitung

Gruppe	≤5	6-10	11-15	≥16	k.A.	Σ
1 – AdlA	26	-	1	-	2	29
2 – GaniFA	26	2	1	-	1	30
3 – Text	30	-	-	-	-	30
4 – Vorlesung	26	2	-	-	1	29
Σ	108	4	2	-	4	118

A7. Andere Anwendungsprogramme

Gruppe	≤5	6-10	11-15	≥16	k.A.	Σ
1 – AdlA	22	2	1	2	2	29
2 – GaniFA	23	5	-	1	1	30
3 – Text	30	-	-	-	-	30
4 – Vorlesung	22	6	-	-	1	29
Σ	97	13	1	3	4	118

A8. Spiele

Gruppe	≤5	6-10	11-15	≥16	k.A.	Σ
1 – AdlA	26	1	-	-	2	29
2 – GaniFA	25	2	-	2	1	30
3 – Text	26	2	2	-	-	30
4 – Vorlesung	24	3	-	1	1	29
Σ	101	8	2	3	4	118

S9. Hast Du schon einmal mit Lernsoftware intensiv gearbeitet?

Gruppe	Ja	Nein	k.A.	Σ
1 – AdlA	7	20	2	29
2 – GaniFA	4	25	1	30
3 – Text	5	25	-	30
4 – Vorlesung	4	24	1	29
Σ	20	94	4	118

Falls ja, zu welchem Themenbereich?

- Fremdsprachen ((4, Gruppe 1), (2, Gruppe 2), (3, Gruppe 3), (2, Gruppe 4))

- Mathematik ((1, Gruppe 1), (2, Gruppe 4))
- Physik (1, Gruppe 3)
- Tutorials (2, Gruppe 1)
- Informatik (1, Gruppe 2)

S10. Wenn Du für das Studium lernst, welche Medien, Hilfsmittel und Informationsquellen benutzt Du? Versuche die Anteile in Prozenten abzuschätzen. (Achtung: Am Ende sollten insgesamt 100 % herauskommen!)

A1. Lehrbücher

Gruppe	≤20 %	21-40 %	41-60 %	61-80 %	≥81 %	k.A.	Σ
1 – AdlA	6	15	5	-	-	3	29
2 – GaniFA	9	15	2	2	1	1	30
3 – Text	11	12	6	-	-	1	30
4 – Vorlesung	13	12	2	1	-	1	29
Σ	39	54	15	3	1	6	118

A2. Skripte

Gruppe	≤20 %	21-40 %	41-60 %	≥61 %	k.A.	Σ
1 – AdlA	9	14	3	-	3	29
2 – GaniFA	9	20	-	-	1	30
3 – Text	10	15	4	-	1	30
4 – Vorlesung	9	15	3	1	1	29
Σ	37	64	10	1	6	118

A3. Übungsblätter

Gruppe	≤20 %	21-40 %	41-60 %	≥61 %	k.A.	Σ
1 – AdlA	9	14	3	-	3	29
2 – GaniFA	13	14	2	-	1	30
3 – Text	9	17	3	-	1	30
4 – Vorlesung	9	15	4	-	1	29
Σ	40	60	12	-	6	118

A4. Internet, WWW

Gruppe	≤5 %	6-10 %	11-15 %	16-20 %	≥21 %	k.A.	Σ
1 – AdlA	13	11	-	2	-	3	29
2 – GaniFA	10	11	2	5	1	1	30
3 – Text	14	10	-	3	2	1	30
4 – Vorlesung	10	11	1	4	2	1	29
Σ	47	43	3	14	5	6	118

A5. Lernsoftware

Gruppe	≤5 %	6-10 %	11-15 %	16-20 %	≥21 %	k.A.	Σ
1 – AdlA	26	-	-	-	-	3	29
2 – GaniFA	28	-	-	-	1	1	30
3 – Text	29	-	-	-	-	1	30
4 – Vorlesung	27	1	-	-	-	1	29
Σ	110	1	-	-	1	6	118

A6. Sonstiges

Gruppe	≤5 %	6-10 %	11-15 %	16-20 %	≥21 %	k.A.	Σ
1 – AdlA	26	-	-	-	-	3	29
2 – GaniFA	26	2	1	-	-	1	30
3 – Text	23	5	-	-	1	1	30
4 – Vorlesung	24	2	1	1	-	1	29
Σ	99	9	2	1	1	6	118

S11. Wie viele Bücher (außer Fachbücher für das Studium) hast Du im letzten Jahr gelesen?

Gruppe	≤5	6-10	11-15	16-20	≥21	k.A.	Σ
1 – AdlA	18	6	1	-	1	3	29
2 – GaniFA	18	9	-	2	-	1	30
3 – Text	16	5	-	6	2	1	30
4 – Vorlesung	19	6	1	2	-	1	29
Σ	71	26	2	10	3	6	118

S12. Wie oft liest Du Tageszeitungen?

A1. Nie
A2. Ca. ein- bis zweimal im Monat
A3. Ca. einmal in der Woche
A4. Ca. zwei- bis dreimal in der Woche
A5. Täglich

Gruppe	A1	A2	A3	A4	A5	k.A.	Σ
1 – AdlA	5	5	3	8	6	2	29
2 – GaniFA	5	5	10	2	8	-	30
3 – Text	3	4	7	5	10	1	30
4 – Vorlesung	4	2	9	6	8	-	29
Σ	17	16	29	21	32	3	118

S13. Hattest Du in der Schule Mathematik als ...

A1. Leistungskurs oder

A2. Grundkurs (Falls ja, bis zu welcher Klasse?)

Gruppe	A1	A2	k.A.	Σ
1 – AdlA	21	6	2	29
2 – GaniFA	20	8	2	30
3 – Text	26	4	-	30
4 – Vorlesung	23	6	-	29
Σ	90	24	4	118

Bei Grundkurs bis zu Klasse

- 12 ((1, Gruppe 1), (1, Gruppe 2))
- 13 ((5, Gruppe 1), (5, Gruppe 2), (5, Gruppe 3), (6, Gruppe 4))

S14. Mit welcher Schulnote [1 bis 6] hast Du den Mathematikkurs abgeschlossen?

Gruppe	1	2	3	4	k.A.	Σ
1 – AdlA	11	9	3	3	3	29
2 – GaniFA	11	14	2	2	1	30
3 – Text	13	8	7	-	2	30
4 – Vorlesung	13	8	5	2	1	29
Σ	48	39	17	7	7	118

S15. Hast Du in der Schule einen Informatikkurs besucht?

A1. Ja, als Leistungskurs (letzte Note bitte angeben)
A2. Ja, als Grundkurs (bis zur Klasse sowie letzte Note bitte angeben)
A3. Ja, als freiwillige Arbeitsgemeinschaft
A4. Nein

Gruppe	A1	A2	A3	A4	k.A.	Σ
1 – AdlA	4	15	2	6	2	29
2 – GaniFA	4	13	1	10	2	30
3 – Text	2	16	4	8	-	30
4 – Vorlesung	1	20	1	6	1	29
Σ	11	64	8	30	5	118

Gruppe	Note 1		Note 2		Note 3		k.A.	Σ
	LK	GK	LK	GK	LK	GK		
1 – AdlA	2	13	2	2	-	-	10	29
2 – GaniFA	3	11	-	2	1	-	13	30
3 – Text	-	13	2	3	-	-	12	30
4 – Vorlesung	1	12	-	6	-	-	10	29
Σ	6	49	4	13	1	-	45	118

Bei Grundkurs bis zu Klasse

- 11 ((1, Gruppe 1), (2, Gruppe 3), (2, Gruppe 4))
- 12 ((2, Gruppe 1), (1, Gruppe 3), (1, Gruppe 4))
- 13 ((13, Gruppe 1), (14, Gruppe 2), (13, Gruppe 3), (15, Gruppe 4))

S16. Hast Du schon einmal an einem Lernexperiment im Rahmen des GANIMAL*-Projektes teilgenommen?*

Gruppe	Ja	Nein	k.A.	Σ
1 – AdlA	-	27	2	29
2 – GaniFA	-	30	-	30
3 – Text	-	30	-	30
4 – Vorlesung	-	29	-	29
Σ	-	116	2	118

Literaturverzeichnis

[Asy02] Asymetrix. ToolBook, 2002. `http://www.asymetrix.com/`.

[Bae73] R. Baecker. Towards Animating Computer Programs: A First Progress Report. In: *Proceedings of the 3. NRC Man-Computer Communications Conference*, 1973.

[Bae81] R. Baecker (mit Assistenz von D. Sherman). Sorting Out Sorting. 30 Minuten Farbfilm (vertrieben durch Morgan Kaufmann Pub.), 1981.

[Bae98] R. Baecker. Sorting Out Sorting: A Case Study of Software Visualization for Teaching Computer Science. In: J. T. Stasko, J. Domingue, M. H. Brown und B. A. Price, Hrsg., *Software Visualization: Programming as a Multimedia Experience*, Kap. 24, S. 369–381. MIT Press, Cambridge, MA, 1998.

[BD02] J. Bortz und N. Döring. *Forschungsmethoden und Evaluation für Human- und Sozialwissenschaftler*. Springer, Berlin, Heidelberg, 3. Auflage, 2002.

[BDK+02] B. Braune, S. Diehl, A. Kerren, T. Weller und R. Wilhelm. Generating Finite Automata – An Interactive Online Textbook, 2002. `http://www.cs.uni-sb.de/GANIMAL/GANIFA`.

[BDKW99] B. Braune, S. Diehl, A. Kerren und R. Wilhelm. Animation of the Generation and Computation of Finite Automata for Learning Software. In: *Automata Implementation, Proceedings of the 4th International Workshop on Implementing Automata (WIA '99)*, Band 2214 der Reihe *Lecture Notes on Computer Science, LNCS*, S. 39–47, Potsdam, 1999. Springer.

[Ber00] R. Berghammer. KIEL: Ein Programm für die Visualisierung der Auswertung von funktionalen Programmen. In: *S. Diehl und A. Kerren, Hrsg.: Tagungsband zum GI Workshop Software Visualisierung (SV '00)*, Mai 2000.

[BH98a] M. H. Brown und J. Hershberger. Fundamental Techniques for Algorithm Animation Displays. In: J. T. Stasko, J. Domingue, M. H. Brown und B. A. Price, Hrsg., *Software Visualization*. MIT Press, 1998.

[BH98b] M. H. Brown und J. Hershberger. Program Auralization. In: J. T. Stasko, J. Domingue, M. H. Brown und B. A. Price, Hrsg., *Software Visualization:*

Programming as a Multimedia Experience, Kap. 10, S. 137–143. MIT Press, 1998.

[BK91] J. L. Bentley und B. W. Kernighan. A System for Algorithm Animation. *Computing Systems*, 4(1), Winter 1991.

[Blu98] A. Blumstengel. *Entwicklung hypermedialer Lernsysteme*. Wissenschaftlicher Verlag Berlin, WVB, 1998.

[BMST02] M. Ben-Ari, N. Myller, E. Sutinen und J. Tarhio. Perspectives on Program Animation with Jeliot. In: *Software Visualization*, Band 2269 der Reihe *LNCS State-of-the-Art Survey*, S. 31–45. Springer, 2002.

[BN93] M. H. Brown und M. A. Najork. Algorithm Animation Using 3D Interactive Graphics. In: *Proceedings of the ACM Symposium on User Interface Software and Technology (UIST '93)*, S. 93–100, Atlanta, GA, USA, November 1993.

[BN96] M. H. Brown und M. Najork. Collaborative Active Textbooks: A Web-Based Algorithm Animation System for an Electronic Classroom. In: *Proceedings of the IEEE International Symposium on Visual Languages (VL '96)*, Boulder, CO, USA, 1996.

[BN98] M. H. Brown und M. A. Najork. Algorithm Animation Using Interactive 3D Graphics. In: J. T. Stasko, J. Domingue, M. H. Brown und B. A. Price, Hrsg., *Software Visualization: Programming as a Multimedia Experience*, Kap. 9, S. 119–135. MIT Press, 1998.

[BN01] M. H. Brown und M. A. Najork. Three-Dimensional Web-Based Algorithm Animations. Technischer Bericht 170, Compaq Systems Research Center, Juli 2001.

[BN02] M. Bäsken und S. Näher. GeoWin: A Generic Tool for Interactive Visualization of Geometric Algorithms. In: *Software Visualization*, Band 2269 der Reihe *LNCS State-of-the-Art Survey*, S. 88–100. Springer, 2002.

[Bor99] J. Bortz. *Statistik für Sozialwissenschaftler*. Springer, Berlin, Heidelberg, 5. Auflage, 1999.

[BP94] P. Baumgartner und S. Payr. *Lernen mit Software*. Digitales Lernen. Österreichischer StudienVerlag, Innsbruck, 1994.

[BR96] M. H. Brown und R. Raisamo. JCAT: Collaborative Active Textbooks Using Java. In: *Proceedings of the Conference CompuGraphics '96*, Paris, Frankreich, 1996.

[Bro87] M. H. Brown. *Algorithm Animation*. MIT Press, 1987.

[Bro88a] M. H. Brown. Exploring Algorithms with Balsa-II. *Computer*, 21(5), 1988.

[Bro88b] M. H. Brown. Perspectives on Algorithm Animation. In: *Proceedings of the ACM Conference on Human Factors in Computing Systems (SIGCHI '88)*, S. 33–38, Washington D.C., Mai 1988.

[Bro91] M. H. Brown. ZEUS: A System for Algorithm Animation and Multi-View Editing. In: *Proceedings of the IEEE Workshop on Visual Languages (VL '91)*, S. 4–9, Kobe, Japan, Oktober 1991.

[Bru61] J. S. Bruner. The Act of Discovery. *Harvard Educational Review*, 31:21–32, 1961.

[BS84] M. H. Brown und R. Sedgewick. A System for Algorithm Animation. In: *Proceedings of ACM Conference SIGGRAPH '84*, Minneapolis, MN, USA, 1984.

[CBC96] P. Carlson, M. Burnett und J. Cadiz. A Seamless Integration of Algorithm Animation into a Declarative Visual Programming Language. In: *Proceedings of the International Workshop on Advanced Visual Interfaces (AVI '96)*, 1996.

[CLK00] A.I. Concepcion, N. Leach und A. Knight. Algorithma 99: An Experiment in Reusability and Component-Based Software Engineering. In: *Proceedings of the 31st ACM Technical Symposium on Computer Science Education (SIGCSE 2000)*, S. 162–166, Austin, TX, USA, Februar 2000.

[Con87] J. Conklin. Hypertext – An Introduction and a Survey. *IEEE Computer*, 20(9):17–41, 1987.

[CR92] K. C. Cox und G.-C. Roman. Abstraction in Algorithm Animation. In: *Proceedings of the IEEE Workshop on Visual Languages (VL '92)*, S. 18–24, Seattle, WA, USA, September 1992.

[CWH99] M. Campione, K. Walrath und A. Huml. *The Java Tutorial Continued*. Addison Wesley, 2. Auflage, 1999.

[DB98] H. L. Dershem und P. Brummund. Tools for Web-based Sorting Animations. In: *Proceedings of the 29th ACM Technical Symposium on Computer Science Education (SIGCSE '98)*, S. 222–226, Atlanta, GA, USA, Februar 1998.

[DETT94] G. Di Battista, P. Eades, R. Tamassia und I. G. Tollis. Algorithms for Drawing Graphs: an Annotated Bibliography. *Computational Geometry: Theory and Applications*, 4:235–282, 1994.

[DF99] C. Demetrescu und I. Finocchi. A Technique for Generating Graphical Abstractions of Program Data Structures. In: *Proceedings of the 3rd International Conference on Visual Information Systems (Visual '99)*, LNCS 1614, S. 785–792, Amsterdam, Niederlande, 1999. Springer.

[DF01] C. Demetrescu und I. Finocchi. Smooth Animation of Algorithms in a Declarative Framework. *Journal of Visual Languages and Computing*, 12(3):253–281, Juni 2001.

[DFS02] C. Demetrescu, I. Finocchi und J. T. Stasko. Specifying Algorithm Visualizations: Interesting Events or State Mapping? In: *Software Visualization*, Band 2269 der Reihe *LNCS State-of-the-Art Survey*, S. 16–30. Springer, 2002.

[DG02] S. Diehl und C. Görg. Graphs, They are Changing – Dynamic Graph Drawing for a Sequence of Graphs. In: *Proceedings of the 10th International Symposium on Graph Drawing*, Irvine, CA, USA, August 2002.

[DGK00] S. Diehl, C. Görg und A. Kerren. Foresighted Graphlayout. Technischer Bericht A/02/2000, FR 6.2 – Informatik, Universität des Saarlandes, Dezember 2000.
`http://www.cs.uni-sb.de/tr/FB14`.

[DGK01] S. Diehl, C. Görg und A. Kerren. Preserving the Mental Map using Foresighted Layout. In: *Proceedings of Joint Eurographics – IEEE TCVG Symposium on Visualization (VisSym '01)*, Eurographics, S. 175–184, Ascona, Schweiz, 2001. Springer.

[DGK02] S. Diehl, C. Görg und A. Kerren. Animating Algorithms Live and Post Mortem. In: *Software Visualization*, Band 2269 der Reihe *LNCS State-of-the-Art Survey*, S. 46–57. Springer, 2002.

[Die02] S. Diehl, Hrsg. *Software Visualization*, Band 2269 der Reihe *LNCS State-of-the-Art Survey*. Springer, 2002.

[DK98] S. Diehl und T. Kunze. Visualizing Principles of Abstract Maschines by Generating Interactive Animations. In: *Proceedings of the Workshop on Principles of Abstract Machines*, Pisa, Italien, 1998.

[DK00a] S. Diehl und A. Kerren. Increasing Explorativity by Generation. In: *Proceedings of the AACE World Conference on Educational Multimedia, Hypermedia and Telecommunications (EDMEDIA '00)*, Montreal, Kanada, 2000. AACE.

[DK00b] S. Diehl und A. Kerren, Hrsg. Tagungsband zum GI Workshop Software Visualisierung (SV '00). Technischer Bericht A/01/2000, FR 6.2 – Informatik, Universität des Saarlandes, Mai 2000.
`http://www.cs.uni-sb.de/tr/FB14`.

[DK00c] S. Diehl und T. Kunze. Visualizing Principles of Abstract Machines by Generating Interactive Animations. *Future Generation Computer Systems*, 16(7), 2000. Elsevier.

[DK01] S. Diehl und A. Kerren. Levels of Exploration. In: *Proceedings of the 32nd ACM Technical Symposium on Computer Science Education (SIGCSE '01)*, S. 60–64, Charlotte, NC, USA, 2001. ACM.

[DK02a] S. Diehl und A. Kerren. Generierung interaktiver Animationen von Berechnungsmodellen. *Informatik – Forschung und Entwicklung*, 17(1):12–20, 2002. Springer.

[DK02b] S. Diehl und A. Kerren. Reification of Program Points for Visual Execution. In: *Proceedings of the First IEEE International Workshop on Visualizing Software for Understanding and Analysis (VisSoft '02)*, S. 100–109, Paris, Frankreich, Juni 2002. IEEE Computing Society Press.

[DKP97] S. Diehl, T. Kunze und A. Placzek. GANIMAM: Generierung interaktiver Animationen von abstrakten Maschinen. In: *Tagungsband zum 3. Fachkongreß Smalltalk und Java in Industrie und Ausbildung (STJA '97)*, S. 185–190, Erfurt, 1997.

[DKW01] S. Diehl, A. Kerren und T. Weller. Visual Exploration of Generation Algorithms for Finite Automata. In: *Implementation and Application of Automata*, Band 2088 der Reihe *Lecture Notes on Computer Science, LNCS*, S. 327–328. Springer, 2001.

[DO02] S. Diehl und M. Ohlmann. InterTalk, 2002. `http://www.cs.uni-sb.de/~diehl/InterTalk/`.

[DS01] R. Douence und M. Südholt. A Generic Reification Technique for Object-Oriented Reflective Languages. *Higher-Order and Symbolic Computation*, 14(1):7–34, 2001.

[Dui87] R. A. Duisberg. Visual Programming of Program Visualizations. A Gestural Interface for Animating Algorithms. In: *Proceedings of the IEEE Computer Society Workshop on Visual Languages (VL '87)*, S. 55–66, Linkøping, Schweden, August 1987.

[EB88] M. Eisenstadt und M. Brayshaw. The Transparent Prolog Machine (TPM): An Execution Model and Graphical Debugger for Logic Programming. *Journal of Logic Programming*, 5(4):1–66, 1988.

[Ede96] W. Edelmann. *Lernpsychologie*. Psychologie Verlags Union, Weinheim, Basel, 5. Auflage, 1996.

[EZ96] P. Eades und K. Zhang, Hrsg. *Software Visualization*. World Scientific Pub., Singapore, 1996.

[Fla00] D. Flanagan. *Java in a Nutshell*. O'Reilly, 3. Auflage, 2000.

[Fou95] S. Foubister. *Graphical Application and Visualisation of Lazy Functional Computation*. Dissertation, Department of Computer Science, University of York, USA, 1995.

[Fra02] J. Françik. Algorithm Animation Using Data Flow Tracing. In: *Software Visualization*, Band 2269 der Reihe *LNCS State-of-the-Art Survey*, S. 73–87. Springer, 2002.

[FW84] D. P. Friedman und M. Wand. Reification: Reflection without Metaphysics. In: *Proceedings of the ACM Conference on LISP and Functional Programming (LFP '84)*, S. 348–355, Austin, TX, USA, 1984. ACM.

[Gan02a] GaniFA. Download Page, 2002. `http://www.cs.uni-sb.de/GANIMAL/download.html`.

[Gan02b] Ganimal. Projekt Homepage, 2002. `http://www.cs.uni-sb.de/GANIMAL`.

[GBW79] R. M. Gagné, L. J. Briggs und R. Wagner. *Principles of Instructional Design*. Holt, Rinehart & Winston, New York, USA, 1979.

[GH97] U. Glowalla und G. Häfele. Einsatz elektronischer Medien: Befunde, Probleme und Perspektiven. In: L. J. Issing und P. Klimsa, Hrsg., *Information und Lernen mit Multimedia*, S. 412–434. Psychologie Verlags Union, Weinheim, Basel, 2. Auflage, 1997.

[GHJV95] E. Gamma, R. Helm, R. Johnson und J. Vlissides. *Design Patterns. Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Addison-Wesley, Reading, Massachusetts, USA, 1995.

[GJSB00] J. Gosling, B. Joy, G. Steele und G. Bracha. *The Java Language Specification*. Sun Microsystems, Inc., 2. Auflage, 2000.

[Glo92] P. A. Gloor. AACE – Algorithm Animation for Computer Science Education. In: *Proceedings of the IEEE Workshop on Visual Languages (VL '92)*, S. 25–31, Seattle, WA, USA, September 1992.

[Glo98a] P. A. Gloor. Animated Algorithms. In: J. T. Stasko, J. Domingue, M. H. Brown und B. A. Price, Hrsg., *Software Visualization: Programming as a Multimedia Experience*, Kap. 27, S. 409–416. MIT Press, Cambridge, MA, USA, 1998.

[Glo98b] P. A. Gloor. User Interface Issues for Algorithm Animation. In: J. T. Stasko, J. Domingue, M. H. Brown und B. A. Price, Hrsg., *Software Visualization: Programming as a Multimedia Experience*, Kap. 11, S. 145–152. MIT Press, Cambridge, MA, USA, 1998.

[GM94] J. Gerstenmaier und H. Mandl. Wissenserwerb unter konstruktivistischer Perspektive. Forschungsbericht Nr. 33, Ludwig-Maximilians-Universität München, März 1994.

[Gör01] C. Görg. Layout animierter Graphen. Diplomarbeit, Universität des Saarlandes, Saarbrücken, 2001.

[GR99] E. Gramond und S. H. Rodger. Using JFLAP to Interact with Theorems in Automata Theory. *SIGCSE Bulletin (ACM Special Interest Group on Computer Science Education)*, 31, 1999.

[H+97] J. Haajanen et al. Animation of User Algorithms on the Web. In: *Proceedings of the IEEE Symposium on Visual Languages (VL '97)*, S. 360–367, Capri, Italien, September 1997.

[Haa97] J. Haack. Interaktivität als Kennzeichen von Multimedia und Hypermedia. In: L. J. Issing und P. Klimsa, Hrsg., *Information und Lernen mit Multimedia*, S. 151–166. Psychologie Verlags Union, Weinheim, Basel, 2. Auflage, 1997.

[HHR89] E. Helttula, A. Hyrskykari und K.-J. Räihä. Graphical Specification of Algorithm Animations with Aladdin. In: *Proceedings of the 22nd Hawaii International Conference on System Sciences (HICSS '89)*, S. 892–901, Kailua-Kona, HI, USA, Januar 1989.

[HMM00] I. Herman, G. Melancon und M. S. Marshall. Graph Visualization and Navigation in Information Visualization: A Survey. *IEEE Transactions on Visualization and Computer Graphics*, 6(1):24–43, 2000.

[Hop74] F. Hopgood. Computer Animation Used as a Tool in Teaching Computer Science. In: *Proceedings of the IFIP Congress*, 1974.

[HR87] A. Hyrskykari und K.-J. Räihä. Animation of Algorithms Without Programming. In: *Proceedings of the IEEE Computer Society Workshop on Visual Languages (VL '87)*, S. 40–54, Linkøping, Schweden, August 1987.

[HU79] J. Hopcroft und J. Ullman. *Introduction to Automata Theory, Languages and Computation.* Addison-Wesley, 1979.

[HWF90] R. R. Henry, K. M. Whaley und B. Forstall. The University of Washington Illustrating Compiler. *Sigplan Notices: SIGPLAN '90*, 25(6):223–233, Juni 1990.

[Ice02] IceSoft Technologies, Inc. IceBrowser Java Bean, 2002. `http://www.icesoft.com`.

[IK97] L. J. Issing und P. Klimsa, Hrsg. *Information und Lernen mit Multimedia.* Psychologie Verlags Union, Weinheim, Basel, 2. Auflage, 1997.

[Iss97] L. J. Issing. Instruktionsdesign für Multimedia. In: L. J. Issing und P. Klimsa, Hrsg., *Information und Lernen mit Multimedia*, S. 195–220. Psychologie Verlags Union, Weinheim, Basel, 2. Auflage, 1997.

[JMM93] D. H. Jonassen, T. Mayes und R. McAleese. A Manifesto for a Constructivist Approach to Uses of Technology in Higher Education. In: T. M. Duffy, J. Lowyck und D. H. Jonassen, Hrsg., *Designing Environments for Constructive Learning*, Band 105 der Reihe *NATO ASI, Series F, Computer and System Sciences*, S. 231–247. Springer, Berlin, Heidelberg, New York, London, 1993.

[KC93] R. A. Knuth und D. J. Cunningham. Tools for Constructivism. In: T. M. Duffy, J. Lowyck und D. H. Jonassen, Hrsg., *Designing Environments for Constructive Learning*, Band 105 der Reihe *NATO ASI, Series F, Computer and System Sciences*, S. 163–188. Springer, Berlin, Heidelberg, New York, London, 1993.

[Ker97] A. Kerren. Animation der semantischen Analyse. Diplomarbeit, Universität des Saarlandes, Saarbrücken, 1997.

[Ker99] A. Kerren. Animation der semantischen Analyse. In: *Tagungsband zur 8. GI Fachtagung Informatik und Schule (INFOS '99)*, Informatik aktuell, S. 108–120. Springer, 1999.

[Ker00] A. Kerren. Visualisierung und Animation der semantischen Analyse von Programmen. *Informatica Didactica – Zeitschrift für fachdidaktische Grundlagen der Informatik*, (1), 2000.

[KH01] S. Khuri und K. Holzapfel. EVEGA: An Educational Visualization Environment for Graph Algorithms. In: *Proceedings of the 6th Annual ACM Conference on Innovation and Technology in Computer Science Education (ITiCSE '01)*. ACM Press, 2001.

[Kli97] P. Klimsa. Multimedia aus psychologischer und didaktischer Sicht. In: L. J. Issing und P. Klimsa, Hrsg., *Information und Lernen mit Multimedia*, S. 6–24. Psychologie Verlags Union, Weinheim, Basel, 2. Auflage, 1997.

[KM96] K. Koskimies und K. Mössenböck. Scene: Using Scenario Diagrams and Active Text for Illustrating Object-Oriented Programs. In: *Proceedings of the 18th IEEE International Conference on Software Engineering (ICSE '96)*, S. 366–375. IEEE Computer Society Press, 1996.

[Kno66] K. Knowlton. L6: Bell Telephone Laboratories Low-Level Linked List Language. 16 Minuten s/w Film, 1966.

[KP97] J. Keller und W. J. Paul. *Hardware Design: Formaler Entwurf digitaler Schaltungen*. Teubner, Stuttgart, 2. Auflage, 1997.

[Kra98] E. Kraemer. Visualizing Concurrent Programs. In: J. T. Stasko, J. Domingue, M. H. Brown und B. A. Price, Hrsg., *Software Visualization: Programming as a Multimedia Experience*, Kap. 17, S. 237–256. MIT Press, Cambridge, MA, USA, 1998.

[KS02] A. Kerren und J. T. Stasko. Algorithm Animation – Chapter Introduction. In: *Software Visualization*, Band 2269 der Reihe *LNCS State-of-the-Art Survey*, S. 1–15. Springer, 2002.

[Kuh91] R. Kuhlen. *Hypertext. Ein nicht-lineares Medium zwischen Buch und Wissenschaft*. Springer, Berlin, Heidelberg, New York, London, 1991.

[Kun99] T. Kunze. Generierung interaktiver Animationen von abstrakten Maschinen. Diplomarbeit, Universität des Saarlandes, Saarbrücken, 1999.

[KWD00] A. Kerren, R. Wilhelm und S. Diehl. MALL – Abschlußbericht, Juni 2000. `http://www.cs.uni-sb.de/RW/projects/mall/`.

[Lan02] H. W. Lang. Heapsort, 2002. `http://www.iti.fh-flensburg.de/lang/algorithmen/sortieren/heap/heap.htm`.

[LBH+97] S. K. Lodha, J. Beahan, T. Heppe, A. Joseph und B. Zane-Ulman. MUSE: A Musical Data Sonification Toolkit. In: *Proceedings of International Conference on Auditory Display (ICAD '97)*, Palo Alto, CA, USA, 1997.

[LD85] R. L. London und R. A. Duisberg. Animating Programs Using Smalltalk. *Computer*, 18(8):61–71, August 1985.

[Leu97] D. Leutner. Adaptivität und Adaptierbarkeit multimedialer Lehr- und Informationssysteme. In: L. J. Issing und P. Klimsa, Hrsg., *Information und Lernen mit Multimedia*, S. 139–149. Psychologie Verlags Union, Weinheim, Basel, 2. Auflage, 1997.

[LM00] J. Lawall und G. Muller. Efficient Incremental Checkpointing of Java Programs. In: *Proceedings of the International Conference on Dependable Systems and Networks (DSN '00)*. IEEE Computer Society Press, 2000.

[MA86] J. S. Milton und J. C. Arnold. *Probability and Statistics in the Engineering and Computer Sciences*. McGraw-Hill, 1986.

[Mac02a] Macromedia. Director, 2002. `http://www.macromedia.com/software/director/`.

[Mac02b] Macromedia. Flash, 2002. `http://www.macromedia.com/software/flash/`.

[MELS95] K. Misue, P. Eades, W. Lai und K. Sugiyama. Layout Adjustment and the Mental Map. *Journal of Visual Languages and Computing*, 6(2):183–210, 1995.

[MGR97] H. Mandl, H. Gruber und A. Renkl. Situiertes Lernen in multimedialen Umgebungen. In: L. J. Issing und P. Klimsa, Hrsg., *Information und Lernen mit Multimedia*, S. 167–178. Psychologie Verlags Union, Weinheim, Basel, 2. Auflage, 1997.

[Mic02] Sun Microsystems. *Java^{TM} Platform Debugger Architecture*, 2002. `http://java.sun.com/products/jpda`.

[MKA90] J. T. Mayes, M. R. Kibby und T. Anderson. Learning About Learning from Hypertext. In: D. H. Jonassen und H. Mandl, Hrsg., *Designing Hypermedia for Learning*, Band 67 der Reihe *NATO ASI, Series F, Computer and System Sciences*, S. 227–250. Springer, Berlin, Heidelberg, New York, London, 1990.

[MPT98] Y. Moses, Z. Polunsky und A. Tal. Algorithm Visualization For Distributed Environments. In: *Proceedings of the IEEE Symposium on Information Visualization (InfoVis '98)*, S. 71–78, 1998.

[MS94] S. Mukherjea und J. T. Stasko. Toward Visual Debugging: Integrating Algorithm Animation Capabilities within a Source Level Debugger. *ACM Transactions on Computer-Human Interaction*, 1(3):215–244, September 1994.

[Naj01] M. A. Najork. Web-Based Algorithm Animation. In: *Proceedings of the 38th Design Automation Conference (DAC '01)*, S. 506–511, Las Vegas, NV, USA, 2001.

[Nap90] T. L. Naps. Algorithm Visualization in Computer Science Laboratories. In: *Proceedings of the 21st ACM Technical Symposium on Computer Science Education (SIGCSE '90)*, S. 105–110, Washington, DC, USA, Februar 1990.

[NEN00] T. L. Naps, J. R. Eagan und L. L. Norton. JHAVE – An Environment to Actively Engage Students in Web-Based Algorithm Visualizations. In: *Proceedings of the 31st ACM Technical Symposium on Computer Science Education (SIGCSE '00)*, S. 109–113, Austin, TX, USA, März 2000.

[Nob02] J. Noble. Visualising Objects: Abstraction, Encapsulation, Aliasing and Ownership. In: *Software Visualization*, Band 2269 der Reihe *LNCS State-of-the-Art Survey*, S. 58–72. Springer, 2002.

[NRW76] D. L. Nelson, V. S. Reed und J. R. Walling. Pictorial Superiority Effect. *Journal of Experimental Psychology: Human Learning and Memory*, 2:523–528, 1976.

[Oes97] B. Oestereich. *Objektorientierte Softwareentwicklung mit der Unified Modeling Language*. Oldenbourg, München, Wien, 3. Auflage, 1997.

[Orn74] R. E. Ornstein. *Die Psychologie des Bewußtseins.* Kiepenheuer und Witsch, Köln, 1974.

[OS02] Rainer Oechsle und Thomas Schmitt. JAVAVIS: Automatic Program Visualization With Object and Sequence Diagrams Using The Java Debug Interface (JDI). In: *Software Visualization*, Band 2269 der Reihe *LNCS State-of-the-Art Survey*, S. 176–190. Springer, 2002.

[OWE96] M. Oudshoorn, H. Widjaja und S. Ellershaw. Aspects and Taxonomy of Program Visualisation. In: P. Eades und K. Zhang, Hrsg., *Software Visualisation.* World Scientific Press, Singapore, 1996.

[Pai86] A. Paivio. *Mental Representations. A Dual Coding Approach.* Oxford University Press, New York, 1986.

[Pap99] C. Pape. *Animation strukturierter Beweise in der universitären Ausbildung.* Dissertation, Universität Karlsruhe, 1999.

[PBS93] B. A. Price, R. Baecker und I. Small. A Principled Taxonomy of Software Visualization. *Journal of Visual Languages and Computing*, 4(3):211–266, 1993.

[PH94] O.-C. Park und R. Hopkins. Dynamic Visual Displays in Media-Based Instruction. *Educational Technology, Research and Development*, 4(34):21–25, 1994.

[PJ98] J. Palsberg und C. B. Jay. The Essence of the Visitor Pattern. In: *Proceedings of the 22nd Annual International Computer Software and Applications Conference (COMPSAC '98)*, S. 9–15, Wien, Österreich, August 1998. IEEE Computer Society Press.

[PN90] S. G. Paris und R. S. Newman. Developmental Aspects of Self-Regulated Learning. *Educational Psychologist*, 25:87–102, 1990.

[PR98] W. C. Pierson und S. H. Rodger. Web-based Animation of Data Structures using JAWAA. In: *Proceedings of the 29th ACM Technical Symposium on Computer Science Education (SIGCSE '98)*, S. 257–260, Atlanta, GA, USA, Februar 1998.

[PS97] C. Pape und P. H. Schmitt. Visualizations for Proof Presentation in Theoretical Computer Science Education. In: Z. Halim, T. Ottmann und Z. Razak, Hrsg., *Proceedings of International Conference on Computers in Education (ICCE '97)*, S. 229–236. AACE - Association for the Advancement of Computing in Education, 1997.

[PSB+94] J. Preece, H. Sharp, D. Benyon, S. Holland und T. Carey. *Human-Computer Interaction.* Addison-Wesley, Wokingham, Reading, Menlo Park, New York, 1994.

[PTW02] J. Palsberg, K. Tao und W. Wang. Java Tree Builder – JTB, 2002. `http://www.cs.purdue.edu/jtb/`.

[Pyt94] M. Pyter. Textrepräsentation in Hypertext. Empirische Analyse von visuellen versus audiovisuellen Sprachdarbietungen in Hypertext. In: *Kongress der DGPs*, Hamburg, 1994.

[RC89] G.-C. Roman und K. C. Cox. A Declarative Approach to Visualizing Concurrent Computations. *Computer*, 22(10):25–36, Oktober 1989.

[RCWP92] G.-C. Roman, K. C. Cox, Donald Wilcox und Jerome Y. Plun. Pavane: a System for Declarative Visualization of Concurrent Computations. *Journal of Visual Languages and Computing*, 3(2):161–193, Juni 1992.

[RF02] G. Rössling und B. Freisleben. ANIMAL: A System for Supporting Multiple Roles in Algorithm Animation. *Journal of Visual Languages and Computing*, 2002.

[Rob98] S. Robbins. The JOTSA Animation Environment. In: *Proceedings of the 31st Hawaii International Conference on Systems Sciences (HICSS '98)*, S. 655–664, Kohala Coast, HI, USA, Januar 1998.

[Rom98] G.-C. Roman. Declarative Visualization. In: J. T. Stasko, J. Domingue, M. H. Brown und B. A. Price, Hrsg., *Software Visualization: Programming as a Multimedia Experience*, Kap. 13, S. 173–186. MIT Press, Cambridge, MA, USA, 1998.

[RS59] M. Rabin und D. Scott. Finite Automata and their Decision Problems. *IBM J. Res. Dev*, 3/2:115–125, 1959.

[RSF00] G. Rössling, M. Schuler und B. Freisleben. The ANIMAL Algorithm Animation Tool. In: *Proceedings of the 5th Annual ACM Conference on Innovation and Technology in Computer Science Education (ITiCSE '00)*, S. 37–40, Helsinki, Finnland, 2000.

[San96] G. Sander. *Visualisierungstechniken für den Compilerbau*. Dissertation, Universität des Saarlandes, Saarbrücken, 1996.

[Sch96] R. Schulmeister. *Grundlagen hypermedialer Lernsysteme*. Addison Wesley, Bonn, 1996.

[Sch00] R. Schulmeister. Zukunftsperspektiven multimedialen Lernens. In: K. H. Bichler und W. Mattauch, Hrsg., *Multimediales Lernen in der medizinischen Ausbildung*. Springer, Heidelberg, 2000.

[SDBP98] J. T. Stasko, J. Domingue, M. H. Brown und B. A. Price. *Software Visualization*. MIT Press, 1998.

[SK93] J. T. Stasko und E. Kraemer. A Methodology for Building Application-Specific Visualizations of Parallel Programs. *Journal of Parallel and Distributed Computing*, 18(2), 1993.

[Ski58] B. F. Skinner. Teaching Machines. *Science*, 128:969–977, 1958.

[SL91] H. Senay und S. G. Lazzeri. Graphical Representation of Logic Programs and Their Behavior. In: *Proceedings of the IEEE Workshop on Visual Languages (VL '91)*, S. 25–31, Kobe, Japan, Oktober 1991.

[SM95] J. T. Stasko und D. S. McCrickard. Real Clock Time Animation Support for Developing Software Visualisations. *Australian Computer Journal*, 27(4):118–128, 1995.

[Sta90a] J. T. Stasko. TANGO: A Framework and System for Algorithm Animation. *Computer*, 23(9):27–39, 1990.

[Sta90b] J. T. Stasko. The Path-Transition Paradigm: A Practical Methodology for Adding Animation to Program Interfaces. *Journal of Visual Languages and Computing*, 1(3):213–236, 1990.

[Sta91] J. T. Stasko. Using Direct Manipulation to Build Algorithm Animations by Demonstration. In: *Proceedings of the ACM Conference on Human Factors in Computing Systems (SIGCHI '91)*, S. 307–314, New Orleans, LA, USA, 1991.

[Sta95] J. T. Stasko. The PARADE Environment for Visualizing Parallel Program Executions: A Progress Report. Technischer Bericht GIT-GVU-95-03, 1995.

[Sta97] J. T. Stasko. Using Student-Built Algorithm Animations as Learning Aids. In: *Proceedings of the 28th ACM Technical Symposium on Computer Science Education (SIGCSE '97)*, San Jose, CA, USA, 1997.

[Sta98] J. T. Stasko. Building Software Visualizations through Direct Manipulation and Demonstration. In: J. T. Stasko, J. Domingue, M. H. Brown und B. A. Price, Hrsg., *Software Visualization: Programming as a Multimedia Experience*, Kap. 14, S. 187–203. MIT Press, Cambridge, MA, 1998.

[Ste95] R. Steinmetz. *Multimedia-Technologie: Einführung und Grundlagen*. Springer, Berlin, Heidelberg, New York, London, 1995.

[STT81] K. Sugiyama, S. Tagawa und M. Toda. Methods for Visual Understanding of Hierarchical Systems. *IEEE Transactions on Systems, Man and Cybernetics*, SMC-11(2):109–125, 1981.

[Sut00] E. Sutinen, Hrsg. *Proceedings of the First Program Visualization Workshop 2000*, Porvoo, Finnland, 2000. Department of Computer Sience, University of Joensuu, Finland.

[SW93] J. T. Stasko und J. F. Wehrli. Three-Dimensional Computation Visualization. In: *Proceedings of the IEEE Symposium on Visual Languages (VL '93)*, S. 100–107, Bergen, Norway, August 1993.

[Szy98] C. Szyperski. *Component Software – Beyond Object-Oriented Programming*. Addison-Wesley, 1998.

[Tal02] A. Y. Tal. Algorithm Animation Systems for Constrained Domains. In: *Software Visualization*, Band 2269 der Reihe *LNCS State-of-the-Art Survey*, S. 101–112. Springer, 2002.

[TD95] A. Y. Tal und D. P. Dobkin. Visualization of Geometric Algorithms. *IEEE Transactions on Visualization and Computer Graphics*, 1(2), 1995.

[Ter97] S.-O. Tergan. Hypertext und Hypermedia: Konzeption, Lernmöglichkeiten, Lernprobleme. In: L. J. Issing und P. Klimsa, Hrsg., *Information und Lernen mit Multimedia*, S. 124–137. Psychologie Verlags Union, Weinheim, Basel, 2. Auflage, 1997.

[ULI02] ULI – Universitärer Lehrverbund Informatik. Projekt Homepage, 2002. `http://www.uli-campus.de`.

[VA96] P. Vickers und J. L. Alty. CAITLIN: A Musical Program Auralisation Tool to Assist Novice Programmers with Debugging. In: *Proceedings of International Conference on Auditory Display (ICAD '96)*, Palo Alto, CA, USA, 1996.

[VA00] P. Vickers und J. L. Alty. Musical Program Auralisation: Empirical Studies. In: *Proceedings of International Conference on Auditory Display (ICAD '00)*, Atlanta, GA, USA, 2000.

[Wal90] J. Walker. A Node-Positioning Algorithm for General Trees. *Software – Practice and Experience*, 20(7):685–705, 1990.

[WC97] J. R. Weinstein und P. R. Cook. FAUST: A Framework for Algorithm Understanding and Sonification Testing. In: *Proceedings of International Conference on Auditory Display (ICAD '97)*, Palo Alto, CA, USA, 1997.

[Web02] WebGain, Inc. Java Compiler Compiler™ – JavaCC, 2002. `http://www.webgain.com/products/java_cc/`.

[Wei96] F. E. Weinert. Lerntheorien und Instruktionsmodelle. In: F. E. Weinert, Hrsg, *Enzyklopädie der Psychologie – Pädagogische Psychologie, Bd. II: Psychologie des Lernens und der Instruktion*, S. 1–48. Hogrefe, Göttingen, 1996.

[Wei97a] B. Weidenmann. Abbilder in Multimedia-Anwendungen. In: L. J. Issing und P. Klimsa, Hrsg., *Information und Lernen mit Multimedia*, S. 107–121. Psychologie Verlags Union, Weinheim, Basel, 2. Auflage, 1997.

[Wei97b] B. Weidenmann. Multicodierung und Multimodalität im Lernprozeß. In: L. J. Issing und P. Klimsa, Hrsg., *Information und Lernen mit Multimedia*, S. 65–84. Psychologie Verlags Union, Weinheim, Basel, 2. Auflage, 1997.

[Wel01] T. Weller. Implementierung und Anwendung der GANIMAL Laufzeitumgebung. Diplomarbeit, Universität des Saarlandes, Saarbrücken, 2001.

[WH95] F. E. Weinert und A. Helmke. Learning from Wise Mother Nature or Big Brother Instructor: The Wrong Choice as Seen from an Educational Perspective. *Educational Psychologist*, 30:135–142, 1995.

[WM96] R. Wilhelm und D. Maurer. *Compiler Design: Theory, Construction, Generation*. Addison-Wesley, 2. Auflage, 1996.

[WS97] R. Watson und E. Salzman. A Trace Browser for a Lazy Functional Language. In: *Proceedings of the 20th Australian Computer Science Conference (ACSC '97)*, S. 356–363, 1997.

[Zub99] H. J. Zuberbühler. Vergleich der Lernwirksamkeit von Animationen und Standbildern. Diplomarbeit, Eidgenössische Technische Hochschule (ETH), Zürich, Schweiz, Oktober 1999.